Analyzing the Conceptual Integrity of Computing Applications Through Ontological Excavation and Analysis

A Dissertation
Presented to
The Academic Faculty

by

Idris Hsi

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
the College of Computing

Georgia Institute of Technology
August 2005

Analyzing the Conceptual Integrity of Computing Applications Through Ontological Excavation and Analysis

Approved by:

Dr. Colin Potts, Advisor
College of Computing
Georgia Institute of Technology

Dr. Spencer Rugaber, Advisor
College of Computing
Georgia Institute of Technology

Dr. James D. Foley
College of Computing
Georgia Institute of Technology

Dr. Leo Mark
College of Computing
Georgia Institute of Technology

Dr. Linda Wills
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: July 18, 2005

“What did you mean it to be made of?” Alice asked, hoping to cheer him up, for the poor Knight seemed quite low-spirited about it.

“It began with blotting-paper,” the Knight answered with a groan.

“That wouldn’t be very nice, I’m afraid – “

“Not very nice *alone*,” he interrupted, quite eagerly: “but you’ve no idea what a difference it makes, mixing it with other things – such as gunpowder and sealing-wax.”

– Lewis Carroll, *Through the Looking-Glass* [43]

I dedicate this dissertation to my parents for their love and support over the many years that it took to produce this work

ACKNOWLEDGEMENT

Few difficult accomplishments were ever finished in isolation. This dissertation has taken me a long time and I can say, with certainty, that without the people named below, I would never have succeeded.

Thank you to the members of the INCITE lab (Institute of Nebulous and Collaborative Intelligent Thought Evolution) for their support in keeping up my morale during the difficult phases of the work and for their suggestions in tackling some of the trickier intellectual problems. I also credit them for providing the necessary prodding at critical moments to take care of the messy logistics of getting the work done (like scheduling a defense). They are Vernard Martin, Heather Richter, Jochen Rick, Eli Tilevich, Patrick Yaner, and Dave Zook. They have been a fine bunch of lab mates and I look forward to future collaborations with them.

Thank you to the faculty members who mentored, guided, and supported me through this work. I especially thank the members of my committee who saw me through the last stages of my PhD work. What I know about research, teaching, and academia, I learned from these professors: Chuck Eastman, Kurt Eiselt, Phil Enslow, Jim Foley, Mary Jean Harrold, Ed Hutchins, Leo Mark, Michael McCracken, Melody Moore, Nancy Nersessian, Russ Shackelford, Linda Wills.

Lastly, I thank my advisors Colin Potts and Spencer Rugaber. Colin picked me up as a student while I was in the process of being exiled from my original department at Georgia Tech. Under his tutelage, I developed my abilities to communicate my ideas, to think deeply about research, and, most important, to identify good research problems. Colin's intellectual fingerprints are scattered throughout this dissertation and it would not

be an exaggeration to say that the work would not exist without his help. In one of my first interactions with Spencer. I was working on a problem on a white board when Spencer stopped in looking for one of the other students in the lab. By the time we had finished chatting, I had five more problems to work on. From Spencer, I learned the value of divergent thinking and methodical validation. Spencer was the one who identified that the core problem that I was trying to solve was that of characterizing and measuring Brooks's idea of conceptual integrity in computing application. He helped me tremendously with removing my angst from the research equation and guiding the logistics for taking this work out of my head and onto paper. Both of my advisors strongly encouraged me to continue this theoretical work whenever I was ready to throw in the towel.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In the world of commercial computing, consumers are being inundated with baroque, bloated, and difficult-to-use *computing applications*, tools that use computational methods and technologies to perform tasks. Market forces demand that new versions of these applications implement more *features*, the user-accessible behaviors and services implemented by the application, than their predecessors or competitors. Ensuring that planned features required for market competitiveness enhance a computing application without these side effects first requires that we understand how these features contribute to the overall design and *conceptual integrity* of the application

While conceptual integrity affects all aspect of the application, we are primarily interested in how an application's user-accessible features have been designed and implemented. To this end, we have developed a research framework, methodologies, and artifacts for measuring the conceptual integrity of a computing artifact from its theory of the world or its *ontology*. We use conceptual coherence, which we define as the degree to which an application's concepts are tightly related, as a first approximation for conceptual integrity.

We claim the following:

- Any computing application has a central or core set of concepts that are essential to that application's ontology and can be identified through analytical means.
- Concepts that are not essential to an application's ontology either exist to support core concepts or are peripheral to the ontology. Peripheral concepts reduce an application's conceptual coherence.

We have developed the method of *ontological excavation* to identify the concepts in a computing application and model them as an ontology expressed as a semantic network. To identify core and peripheral concepts and to measure an ontology's conceptual coherence, we developed methodologies for *ontological analysis*.

If usefulness depends on the conceptual integrity of an application's ontology such that it ensures high fitness to a problem domain, then we would expect that users solving problems in that domain will invoke the concepts integral to the solution more often than those concepts that do not. Thus, to validate our structural measures, we claim the following:

- The probable use of the application will invoke core concepts more frequently than peripheral concepts in the ontology.

1 INTRODUCTION

In the world of commercial computing, consumers are being inundated with baroque, bloated, and difficult-to-use *computing applications*[1], tools that use computational methods and technologies to perform tasks. Market forces demand that new versions of these applications implement more *features*, the user-accessible behaviors and services implemented by the application, than their predecessors or competitors. However, additional features often produce interaction problems with existing features and decrease the usability of the systems. Ensuring that planned features required for market competitiveness enhance a computing application without these side effects requires that we understand how these features contribute to the overall design and *conceptual integrity* of the application, a term used by Fred Brooks in his book, *The Mythical-Man Month* [34]. However, Brooks never offers a concrete definition or characterization of this important design aspect that could be used to guide design.

In this chapter, we present and motivate our research framework for analyzing conceptual integrity in computing applications. We first discuss how Brooks describes conceptual integrity and its importance to design and how he uses building architecture to motivate its importance. We then propose that conceptual integrity originates from an application's *ontology*: its collection of concepts and relationship that embody theories and knowledge about the problem domain that it has been engineered to solve. We argue that an application's *usefulness*, the extent to which an application succeeds in assisting

[1] We introduce a number of terms in this document. A glossary is provided in Appendix A.

users to achieve their goals, directly depends on the conceptual integrity of its ontology. Finally, we present the central claims of this dissertation.

1.1 Conceptual Integrity

Brooks observes that conceptual integrity exists in a system possessing qualities that could only have emerged from a unified vision of that system. A computing application designed with conceptual integrity possesses a software architecture, user interface, and functionality that are easy to comprehend, maintain, and use – qualities so vital to a system's performance and success that Brooks argues the following:

> "I will contend that conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas." *[34]*

To illustrate the idea of conceptual integrity, he uses the Reims cathedral in France as an example of a structure with such conceptual integrity that it evokes joy in the beholder. The Cathédral Notre-Dame de Reims was completed at the end of the 13th century. Its western façade was completed in the 14th century (see Figure 1) [196]. Brooks notes that cathedrals, taking centuries to build, often drifted from their original concept as new generations of designers and builders sought to add their influence to the edifice. This is quite evident in the Cathédral Notre-Dame de Chartres (see Figure 1). A fire in 1194 had destroyed all but the west front. It was rebuilt between 1194 and 1220. In 1506, lightning destroyed the north spire. The results of the two repairs produced a newer, taller tower in a 16th-century Gothic style, looking markedly incongruent to the older tower built in the 12th century Romanesque style [195]. In contrast, the cathedral at Reims possesses an architectural unity and design integrity that could only have been

achieved “by the self-abnegation of eight generations of builders, each of whom sacrificed some of his ideas so that the whole might be of pure design [34].”

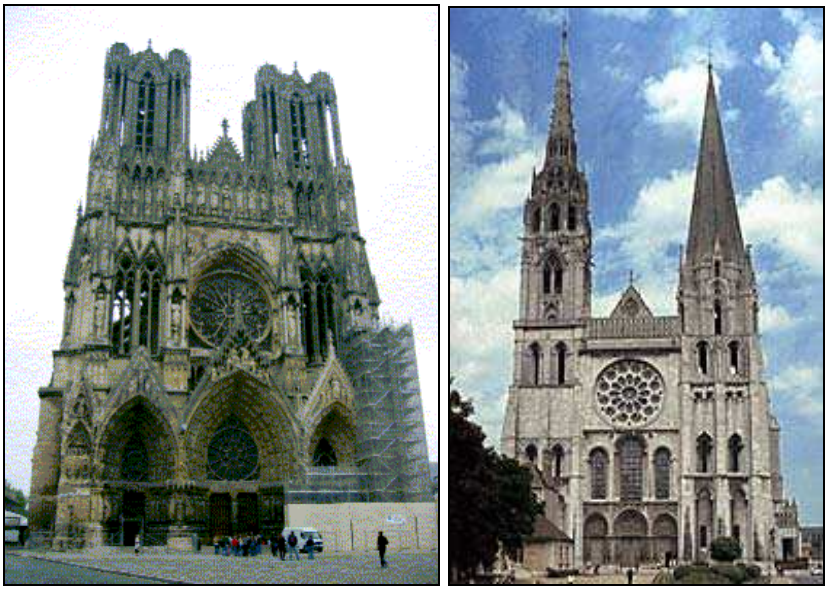


Figure 1 – Notre-Dame de Reims, France (left) [196] and Notre-Dame de Chartres, France (right) [68]

In architecture, the presence of conceptual integrity in a building’s design indicates the extent to which the designer unified the building’s purpose or concept with the constraints of structure and material. One of the earliest writings on this topic comes from the Roman architect Vitruvius who described a ‘Vitruvian triad’ of ‘firmness, commodity, and delight’ that should be present in the design of any structure [90]. Firmness characterizes a building’s ability to withstand external forces, such as weather and fire. Commodity grants its inhabitants comfort and efficiency. Delight endows a building with an aesthetic sensibility that makes a building worth inhabiting beyond its basic functions [172]. This triad describes for architecture what Brooks’s conceptual integrity represents for computing applications – a desirable property in a design.

Buildings and cities are designed for a basic purpose: to enable their inhabitants to live and work in them. Computing applications have many different purposes, but they still require coherent designs to perform their functions well. Vitruvius's terms offer an analogy that could be applied to the computing domain. However, buildings and cities have physical structures and affordances which determine their form and function. In architecture, these concepts of form and function are intertwined, each informing the other. A computing application exists in a virtual form, operating under constraints specified by an artificial environment. Lacking concrete constraints, an application's form and function have no interdependencies. Form, expressed through its user interface, does not necessarily inform function because much of the function may be invisible to the user. Likewise, while an application's functions can influence its form, they do not always inform the application's affordances or appearance. These external controls and displays are determined by designers using accepted standards and designs for interface behavior and usability. Because of the virtual nature of computing and the lack of hard constraints on form and function, the conceptual integrity of a computing application can be much harder to perceive, design, engineer, and characterize.

1.2 Design in Computing

While the discipline of computer science still lacks its own Vitruvian triad and tradition to match those in architecture, there have been many efforts to characterize and assess design in the specific aspects of computing applications. In architecture, Alexander observed coherence and integrity in smaller integrated units of landscape and architectural features that he cataloged into what he called a pattern language [1, 2], which later became the inspiration for design patterns in software architectures [70].

Design attributes that suggest similar notions to conceptual integrity have been qualified in items like code [133, 134] and the "bad smells" judgments used to identify awkward code [67], open source systems such as Linux [169], user interfaces [34, 58, 154], and web pages [190]. While these efforts have helped to inform and improve the development of those specific aspects and components of computing applications, we still lack a unified understanding of conceptual integrity. We believe that the key to understanding and measuring the effects of conceptual integrity lies in understanding the fundamental real world concepts that have been engineered and operationalized into computing applications.

Brooks claims that software systems often reflect conceptual disunity from the separation of design into tasks delegated to different people and, to a lesser extent, from the participation of multiple master designers. In addition to multiple designers, a computing application's design undergoes frequent changes over its lifetime. Adding features to systems can affect an application's conceptual integrity as the process of engineering and integrating these features necessarily alters the underlying design. Thus, to understand conceptual integrity, we must begin by understanding the motivation for designing features into an application.

1.3 Problem Domains and Software Ontologies

Applications are engineered to solve problems in specific *use contexts*. A use context consists of the external physical (or virtual) environment that contains the computing application and its users, the goals that the combined computing application/user system wishes to achieve, and the various factors (business rules, customer demands, user and system capabilities) that govern the operation and performance of both the environment

and the completion of those goals. For example, the use context of a bank customer database consists of the bank itself, the systems that manage and store the database, the employees charged with maintaining the stored information, and the rules and procedures established by bank management for storing and distributing the data.

All use contexts exist to fulfill specific goals within a problem domain. Arango and Prieto-Díaz state that a *problem domain* is a collection of items of real-world information that has "deep or comprehensive relationships among the items of information" and a community that has a stake in solving those problems [9]. Software that has been designed to function in the use context and the problem domain will possess a set of concepts and relationships that we call an *ontology* [32, 65, 78, 140, 192, 193].

The ontology of a computing application is its theory of the real world. For example, a word processor has been engineered with a theory about what documents are and how they are composed; a photo editor has been engineered with a theory about what digital photographs are and how they can be manipulated. The concepts that compose the ontology determine and structure the application's features. Ultimately, users evaluate a computing application by whether its features enable them to achieve their goals, whether these goals concern entertainment, productivity, or learning. Applications that best serve the users are considered useful and can succeed financially. Thus, software engineers should be concerned with ensuring that their products have a high level of usefulness.

1.4 Usefulness

We define *usefulness* as the extent to which an application succeeds in assisting a set of users to achieve a set of goals, relative to the amount of effort required to engage those features. We distinguish usefulness from *usability*, which we define as the amount of

effort required to engage a feature that achieves a useful results. Usability is an integral but subordinate attribute of usefulness. A useful application with poor usability can still enable users to achieve their goals, albeit with great difficulty. An application with little or no usefulness can be extremely usable but cannot help users to achieve their goals.

Developing useful computing applications requires that developers understand what their users are trying to do in a specific use context and encode that knowledge into the design. Yet an application must possess enough features to be useful to its users without becoming too complex – a design tradeoff between functional power and conciseness. Features are accessed by users of the application through its user interface, which is its external presentation.

The features must ultimately aid the users of the application to achieve goals in the problem domain of the use context. Thus, what the application is, how it is presented to the users, and how it functions must ultimately be determined by its ontology. If its ontology does not match the user's understanding of the problem domain then the application will fail. If the ontology has been modeled correctly, relative to the problem domain, then its concepts will have a high correspondence to parallel concepts in the domain. In other words, the usefulness of a computing application is determined by the *conceptual fitness* of its ontology to the use context. If the ontology lacks conceptual fitness, the most advanced techniques in program design, development, and testing will not produce a useful computing application.

A method for measuring the conceptual fitness of an application's ontology to a use context would allow us to measure the actual and potential usefulness of an application, possibly prior to development. However, measuring conceptual fitness requires both a

comprehensive model of the application's ontology as well as an equivalent and comparable model of its use domain.

1.5 The Research Framework

We have stated the following:

- Conceptual integrity is a desirable quality in computing applications and is evidenced by a well-designed software architecture, user interface, and feature set.
- Computing applications are developed to be useful to their users and to function in specific problem domains.
- These applications embody an ontology – a set of concepts and relationships derived from the problem domain.
- The concepts in the ontology determine what features the software implements.
- The degree to which the ontology matches the problem domain of the use context is its conceptual fitness.
- An application possessing high conceptual fitness is more likely to be useful than one with low conceptual fitness.
- All functional and user-accessible elements of a computing application are based on its ontology.
- Thus, the ontology is the single most important factor determining the conceptual integrity of the application.

The quality of conceptual integrity encompasses much more than the application's ontology and includes other aspects of the application such as its architecture, user interface, and functionality. However, as we argue that these other aspects must necessarily be derived from the ontology, we must first ask ourselves what an ontology with conceptual integrity exhibits in its design, structure, or composition. The answer can be derived from the idea of *conceptual coherence*.

1.6 Conceptual Coherence

We have identified a property of an ontology's conceptual integrity that we call conceptual coherence. Conceptual coherence measures the degree to which a computing application's concepts are tightly related. An ontology oriented around a single main idea is completely coherent. One that contains many concepts that are unrelated is incoherent. Therefore, in order for a computing application to have a high conceptual integrity, it must first possess an ontology with a high conceptual coherence.

Problem domains, such as meeting scheduling, banking, or telephony, have a set of concepts that define them. For example, meeting requires participants, a scheduling procedure, and a reason to meet. Banking involves financial transactions, customers, and accounts. Telephony offers communication services through specific media to connect people to each other. However, some domain descriptions for specific use contexts include seemingly optional concepts. For example, an alarm that reminds the user of an impending meeting might be helpful but may or may not belong to the defining set of concepts that articulate a meeting scheduling domain. Banks may offer investment advice to their customers, something that may or may not be a central concept in banking. Telephony services can include features like vanity numbers or opinion poll numbers [198], which go beyond basic call connections. These supplementary concepts are one step removed from the defining set of concepts. If a meeting scheduling application incorporated types of meetings, such as birthdays or holidays, thus encompassing the broader category of event scheduling, is its ontology still one of meeting scheduling? If banking services began to include advice about real estate and home ownership or insurance, is the bank's ontology still strictly about banking? Some cell phones now

include digital cameras in their feature sets that enable them to take pictures and send them to a recipient. Is this still telephony? Since technology evolves with the requirements of its users, these services or the systems implementing these services are still exhibiting conceptual fitness by maintaining correspondences between their ontologies and their users' problem domains. However, an ontology that has added many concepts that are not directly related to its essential and foundational concepts has lost conceptual integrity. Such an ontology no longer expresses a single, unified idea but multiple, disparate ideas not necessarily related to one another.

1.7 Dissertation Thesis

In this dissertation, we claim the following:

- Any computing application has a central or core set of concepts that are essential to that application's ontology and can be identified through analytical means.
- Concepts that are not essential to an application's ontology either exist to support core concepts or are peripheral to the ontology. Peripheral concepts reduce an application's conceptual coherence.

To investigate these, we have developed the method of *ontological excavation* to identify the concepts in a computing application and model them as an ontology expressed as a semantic network. To identify core and peripheral concepts and to measure an ontology's conceptual coherence, we developed methodologies for *ontological analysis*.

We present our methods for ontological excavation in Chapters 2, 3, and 4 and motivate them with examples from our case studies of real applications. In Chapter 5, we discuss our methods for ontological analysis and how we use them on the structure of the semantic network to identify core concepts and conceptual subgroups. We also present

our metric for measuring the conceptual coherence of an ontology and demonstrate, by example, how removing core concepts can reduce this coherence and how removing peripheral ones can increase it.

Usefulness and conceptual integrity are intertwined concepts. However, our methods for modeling and analyzing the ontology are structural and mathematically abstract, lacking direct correspondence to real world behaviors. If usefulness depends on the conceptual integrity of an application's ontology such that it ensures high fitness to a problem domain, then we would expect that users solving problems in that domain will invoke the concepts integral to the solution more often than those concepts that do not. Thus, to validate our structural measures, we claim the following:

- Usage of the application will invoke core concepts more frequently than peripheral concepts in the ontology.

To test this claim, we developed a method that we call *use case silhouetting*, which measures the amount of ontological coverage of a set of use cases for an application. We describe use case silhouetting and its relationship to usefulness in Chapter 6. Figure 2 shows the overview of how we excavate an application's ontology, analyze it, and measure its usefulness using use case silhouetting. It shows the methods (in boxes) that we use and artifacts (in ovals) that we produce, and the sequence of steps that we perform.

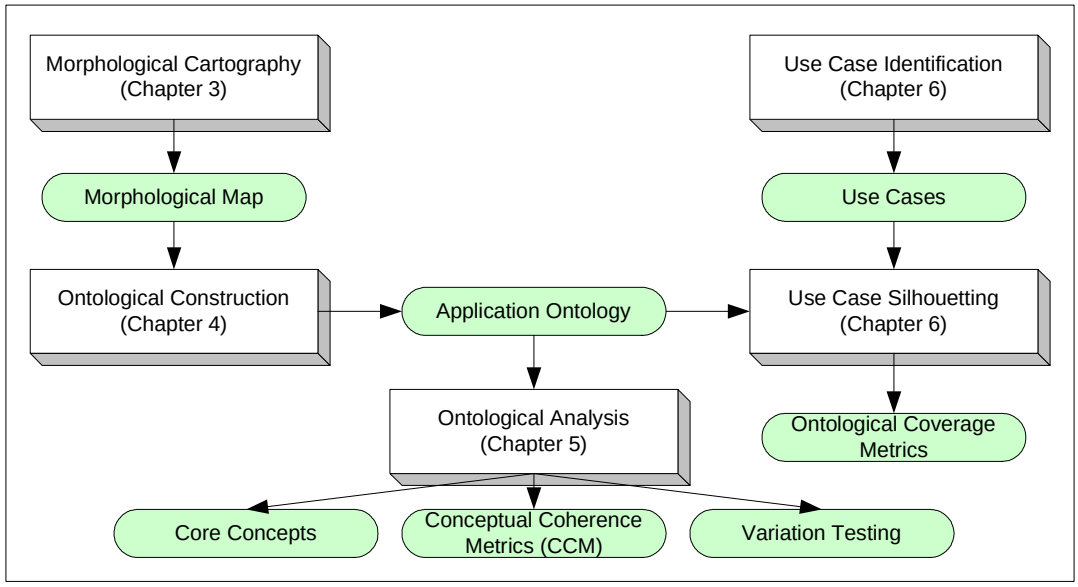


Figure 2 – Overview of Research Methodologies and Artifacts

In Chapter 7, we present a case study using a large application – Microsoft PowerPoint 2000 – and show how our methods scale to large systems. In Chapter 8, we present a constrained excavation of Microsoft Word 2000 and validate our methods on independently gathered usability data. In Chapter 9, we discuss the implications of our findings to an application's evolution across versions. We then discuss the results of our studies in Chapter 10 and account for potential threats to validity in our methods and analyses. In Chapter 1, we propose our future work in this area. Finally, we conclude with some thoughts on our contributions in Chapter 12.

2 BLACK BOX REVERSE ENGINEERING AND ONTOLOGICAL EXCAVATION

We wish to characterize and analyze the conceptual integrity of applications from the perspective of their conceptual coherence and usefulness. To do this, we must first examine those concepts in an application's ontology that compose its embodied theory of the world. In this section, we present our overview of *ontological excavation*, a black-box reverse engineering technique for identifying domain concepts and relationships from an application's features. We first discuss our rationale for using black-box methods. We then review the basic steps in ontological excavation. Next, we introduce the case study applications we use to demonstrate our methods. Lastly, we summarize the software applications used in this work to build our models of the computing application morphologies and ontologies and to develop visualizations for these models and for our analyses.

2.1 Black Box Reverse Engineering

2.1.1 Program Comprehension and Reverse Engineering

Our method for ontological excavation is an activity of program comprehension – "the process of acquiring knowledge about a computer program [173]." Specifically, our excavation methods are a type of reverse engineering defined by Chikofsky and Cross as "the process of analyzing a subject system to identify the system's components and their interrelationships and create representations of the system in another form or at a higher level of abstraction. [46]" Rugaber describes five gaps that complicate the conceptual understanding of programs [173]:

- The gap between a problem from some application domain and a solution to it in some programming language.
- The gap between the concrete world of physical machines and computer programs and the abstract world of high-level descriptions.
- The gap between the desired coherent and highly structured description of a system as originally envisioned by its designers and the actual system whose structure may have disintegrated over time.
- The gap between the hierarchical world of programs and the associational nature of human cognition.
- The gap between the bottom-up analysis of source code and the top-down synthesis of the description of the application.

Ontological excavation addresses the first three of these gaps but only indirectly as we use black-box reverse engineering methods to reverse engineer the domain of the system. White-box methods for reverse-engineering use processes that analyze the source code of a program. The metaphor of a white box suggests the visibility of underlying functionality – system-level operations that enable user- and system-level features can be viewed and studied. Conversely, black-box methods are used when the application is opaque – the system-level operations are inaccessible and only the user-perceivable behaviors and states can be observed.

In domain analysis and reverse engineering, researchers have developed methods for extracting the domain from program documentation [3], requirements specifications [74], source code [47, 56], and interviews with domain experts [8]. Of these techniques, source code analysis has the most rigorous tools for automation, thorough identification of system concepts, and structured analysis. However, source code contains numerous

concepts that concern the management and coordination of the computing operations and hardware interactions that implement the features. While the encoding and interactions of these system concepts do contribute to the application's overall conceptual integrity, we need to distinguish them from the problem domain concepts embedded in the user-level features. Therefore, we designed ontological excavation to use black-box reverse engineering methods.

2.1.2 Black-box Reverse Engineering

Black-box reverse engineering follows the tradition of other qualitative research methods, such as those in ecological psychology, ethnography, and cognitive anthropology. These methods are designed to develop an understanding of a subject's domain by analyzing the use of language and artifacts in the subject's environment. These contextual or naturalistic approaches, unlike abstractionist or laboratory approaches, take the perspective that human behavior is grounded in an environment and context, and that these behaviors can only be understood in that context [100, 104, 148, 149]. These methods also stress that they be performed without preconceived notions about what will be discovered [128, 181]. Our black-box reverse engineering techniques are grounded in this tradition. By treating the computing application as the unit of analysis and referencing domain knowledge directly from the application whenever possible, we can construct the ontology of the problem domain as it has been encoded in the application.

2.2 The Ontological Excavation Process

We excavate the user-level concepts of the ontology from the morphology of the computing application [96]. We define the *morphology* of a computer application as the

external presentation of a computing application consisting of those elements that are both user-accessible and perceivable. In a command-line user interface, the morphology consists of the display and all the commands and their variations that can be entered at the user prompt. In a graphical user interface, the morphology consists of windows and interface widgets, such as buttons and text fields. In a device or appliance, the morphology consists of physical instruments, such as displays, buttons, dials, and switches that activate functionality.

The excavation metaphor comes from archaeology or geology where researchers or engineers have to dig through an exterior layer of earth or rock to uncover items of interest. In our case, *ontological excavation* is the process of digging through an application's exterior surface – its morphology – to identify the domain concepts that compose the application's ontology. The components comprising the morphology are portals to the underlying ontology, each revealing specific concepts and relationships. Through systematic interaction with the application's outer shell, we identify or "excavate" the concepts and the basic relationships between those concepts to model the ontology into a semantic network.

In summary, the basic steps of ontological excavation are:

1. Morphological cartography – Model the application morphology into a morphological map.
2. Ontological construction – Identify the concepts and relationships and model them as a semantic network.

We explain morphological cartography in detail in Chapters 3. Ontological construction is described in Chapter 4.

2.3 Overview of Case Study Applications

To demonstrate our methods for ontological excavation, we include examples from case studies we conducted on real world computing applications [93]: The Microsoft Windows 95/98 CD Player, The Palm Pilot 2000 Scheduler, Microsoft Notepad, and Protocol's Calendar / Calculator.

2.3.1 The Microsoft Windows 95/98 CD Player

The CD Player (Figure 3) allows the user to play CDs, to manage information about that CD, which has to be entered manually by the user, and to manage custom playlists.

Figure 3 – The Win 95/98 CD Player

2.3.2 The Palm Pilot 2000 Scheduler

The Scheduler ran on the Palm Pilot 2000 and provided its user with features such as event scheduling, alarms, and synchronization with other applications.

2.3.3 Protocol's Calendar / Calculator

The Calendar / Calculator (Figure 4) is a device that implements an alarm clock, a calendar, a calculator, a currency exchange calculator, and a countdown timer. The clock also allows its users to view times in sixteen different time zones.

Figure 4 – Protocol's Calendar / Calculator – Exterior (left), Interior (right)

2.3.4 Microsoft Notepad

MS Notepad (Figure 5) is a text editor that comes with the Windows operating systems. Notepad accepts a variety of types of text files in different encodings and displays and prints a document using application settings that are applied to every text file read by Notepad. These display and print settings are not saved by the program.

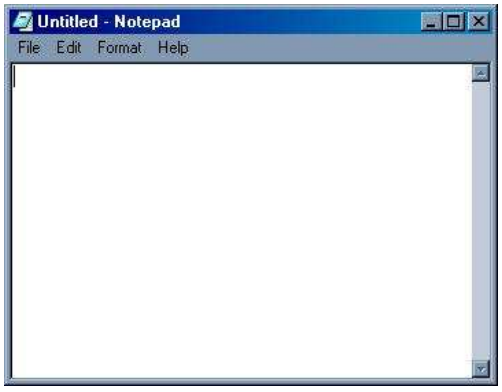


Figure 5 – MS Notepad application

2.4 Overview of Modeling and Visualization Tools

We use a number of software applications to build our morphology and ontology models. We also use these applications to develop visualizations of our models and our analyses. In this section, we review these tools and our general procedures for using them.

2.4.1 Modeling Tools

We model the application morphologies and ontologies using Microsoft Visio 2002, a drawing tool. We wrote a macro to process and save the morphology and ontology graphs as adjacency list representations stored in DL-format files. Both the macro and file format can be found in Appendix C. We then analyze these adjacency lists using a social network analysis tool called UCINET [29], which has a number of algorithms for processing graphs and for calculating various graph properties. As part of our analysis, we also process and sort the data gathered from UCINET's network metrics and graph algorithms on a spreadsheet in Microsoft Excel. We discuss the procedures for ontological analysis in Chapter 5.

2.4.2 Visualization Tools

Occasionally, our morphological and ontological modeling produces large graphs than cannot be reasonably displayed in a document, even with moderate resizing. To display this data and to supplement some of our analyses and conclusions, we include visualizations of graphs representing morphologies and ontologies of the applications that we studied. To produce these visualizations, we import our DL-format files into a social network visualization program called NetDraw [27]. NetDraw has features for calculating and displaying graphs using various user-specified parameters, and it was used to generate our 2-D visualizations.

We also use NetDraw to generate 3-D visualizations. NetDraw has a feature that applies a spring-embedding algorithm to the graph. This algorithm sets the length of an edge between two nodes as a function of the other nodes in the neighborhood. NetDraw saves the modified 2-D graph as a 3-D KineMAGE. MAGE is a program for visualizing organic molecules and protein configurations developed by David Richardson through the Biochemistry Department at Duke University [170].

Throughout this document, we use these visualizations to reinforce and present conclusions that we have obtained using other analytical methods. It is important to note that the visualizations themselves, and the process by which they were obtained (e.g. the algorithms), are not central contributions to this work.

3 MORPHOLOGICAL CARTOGRAPHY: GENERATING THE MORPHOLOGICAL MAP

In this chapter, we present both the methods and the representation used to model the *morphology* of a computing application as a graph that we call a *morphological map*. We describe our abstract framework of *morphological elements*, things that structure and function in a morphology. We discuss our conventions for modeling the interactions and relationships between these elements. We present the heuristics we use to traverse the morphologies of our examples. Lastly, we consider some previous work in the area of user interface reverse-engineering and the issues we identified that can affect future work to automate this process.

3.1 The Morphological Map

In keeping with the archaeological metaphor, we call our methodology for developing the model of a computing application's morphology, *morphological cartography*. Black-box methods can be imprecise if not applied methodically and with some external references to ensure complete coverage of the morphology. Thus, the first task must be to survey the external elements of the computing application to develop a comprehensive "map" of the morphology. In this work, we primarily consider those computing applications with graphical user interfaces, although, as we show with our Calendar / Calculator case study, our methods can be applied to other types of interfaces as well.

We model the *morphological map* as a connected graph of morphological elements and connections. *Morphological elements* are any visible and distinct component in the

morphology that allow interaction with the system, display state, or structure the morphology. *Connections* are arrows that show how the elements are accessed in the morphology by a user and how elements structure other elements. We build the map by traversing and activating all the user interface elements in a systematic, depth-first fashion. These elements, their labels, and their connections are drawn and assembled in a Microsoft Visio diagram. Currently, modeling the user interface as a morphological map is performed manually.

3.2 Overview of Morphological Cartography

The basic steps of morphological cartography are:

1. Survey the morphology – Systematically search for and identify the major structuring elements (containers) in the morphology.
2. Identify morphological components – For each identifiable component within a container, determine which category it belongs to: container, interactor, display, or interactive object.
3. Diagram component in map – Insert the appropriate symbol and name for the object in the map.
4. Diagram containment – Draw an arrow from the parent container to the contained component to indicate possession in the map.
5. Diagram navigation – Use each interactor or interactive object, where appropriate, to determine whether their behaviors change the state of the morphology (e.g. does pushing a button pop up a dialog box, change modes of the system, change a display?). If so draw an arrow from the interactor to the new or existing container to indicate a navigation behavior.

3.3 Step 1: Survey the Morphology

Before any kind of mapping activity, the landscape must first be surveyed to ensure that all important features have been identified for inclusion in the map. Surveying the morphology produces a strategy for ensuring complete or near-complete coverage of the user-accessible components. Generally, surveying is a combination of visually scanning the main morphology and interacting with some of the components to gauge the depth and breadth of the morphology.

In virtual morphologies, such as graphical user interfaces on desktop computers, and physical morphologies, such as the instrument panel for a device, a top-down, depth-first strategy can provide reasonable coverage. However, applications with different modes of operation or entangled navigation between morphological elements may require multiple recursive passes on all the elements to ensure complete coverage.

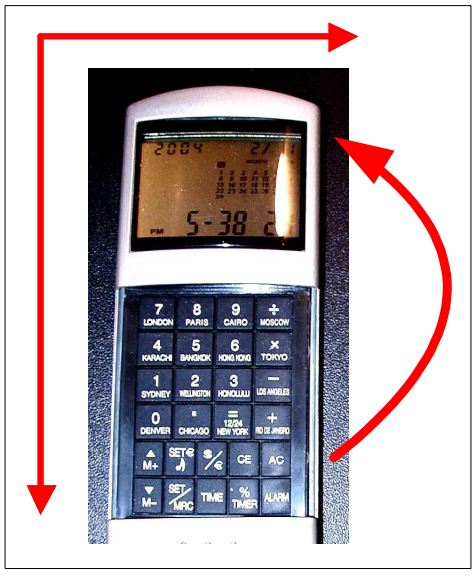


Figure 6 – Recursive traversal strategy for Calculator / Calendar morphology

For example, Figure 6 shows a very simple left-right, top-down survey of the Calculator / Calendar morphology. Starting with the display then identifying all the keys and their different functions seems a reasonable way to start. However, many of the keys have multiple functions. The “7” key serves as a quick check on the London Time Zone in the normal Time mode, as well as the way to input the number 7 in the Calculator or Currency Exchange Calculator modes. Because the Calculator / Calendar has different modes, the strategy is to recursively traverse the device when a new mode has been identified to determine which buttons are used in that mode. Buttons that activate the different modes are located at the bottom of the device, so as the arrow shows, when a new mode is entered, traversal recursively begins from the top of the device to the bottom, moving left to right.

3.4 Step 2: Identifying and Categorizing Morphological Elements

Each morphological element is represented by a visual icon and given information corresponding to its label in the user interface. We abstracted our morphological elements taxonomy from the Swing component framework in Java [184] and from the Foley hierarchy [66, 145]. Besides the standard graphical user interface (GUI) representations [58, 92], there are a variety of alternative user interface representation techniques that include state transition networks, application frameworks, and context-free grammars [145, 147]. We identified a minimal set of abstract categories that allowed sufficient detail to model the morphology of any computing application. Our categories for morphological elements are:

- *Interactors* – user-activated elements that perform operations in the application (e.g. buttons, text fields, checkboxes)

- *Containers* – elements that contain and structure interactors (e.g. windows, dialog boxes, toolbars)
- *Displays* – elements that passively present either static or dynamic data about the computing application's states to the user (e.g. text displays, status bars).
- *Interactive Objects* – objects whose properties can be modified by the user and/or exhibit independent behaviors and states (e.g. embedded objects, work products, virtual worlds, game objects).

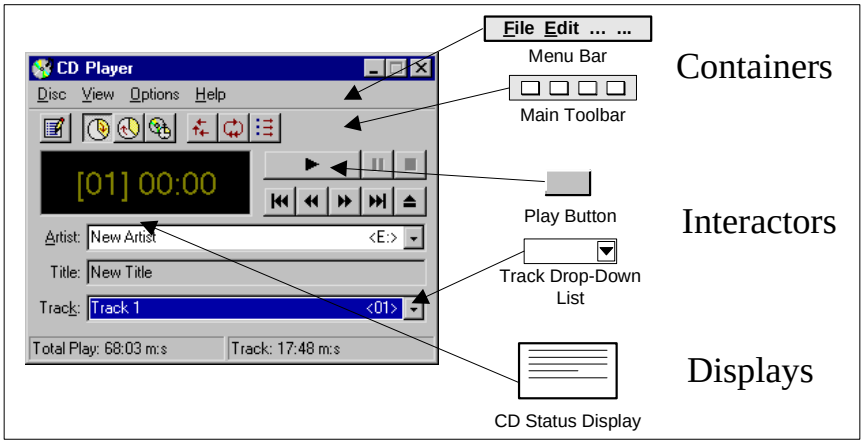


Figure 7 – Examples of containers, interactors, and displays from the CD Player

Figure 7 shows a screenshot of the CD Player along with examples of containers, interactors and displays. Each element has a symbol to identify it in a map. The specific items, such as "drop-down list" or "menu bar" are standard names for these elements in user interface design.

3.5 Step 3: Diagram Components In Map

We diagram the components identified in the morphology using symbols and label them using as much information from the morphology as possible. This information consists mainly of text associated with the component, in the form of labels on or next to the item. When such information is insufficient, we resort to our own terminology, but

use brackets, “[ ]”, to highlight where we have substituted our own inferences (explained further in 3.7). We also identify the elements and their containment using abbreviations. Conventions and abbreviations for morphological mapping are described in Appendix D.

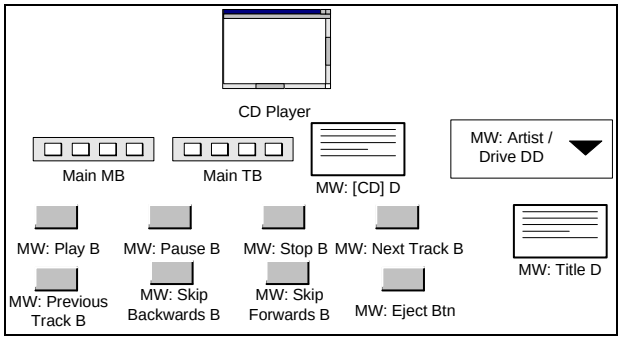


Figure 8 – Morphological elements from the CD Player

Figure 8 shows some of the CD Player’s morphological elements. The item labeled “MW: [CD] D” is a display (D) for the CD (inferred) contained in the application’s main window (MW). “

3.6 Steps 4 and 5: Diagram Containment and Navigation

Elements in the map are linked using arrows to show either their container (e.g. a toolbar containing buttons) or their point of activation (e.g. a menu item opening a dialogue box). Figure 9 shows how the menu bar is modeled in the CD Player morphological map and as an example of containment arrows.

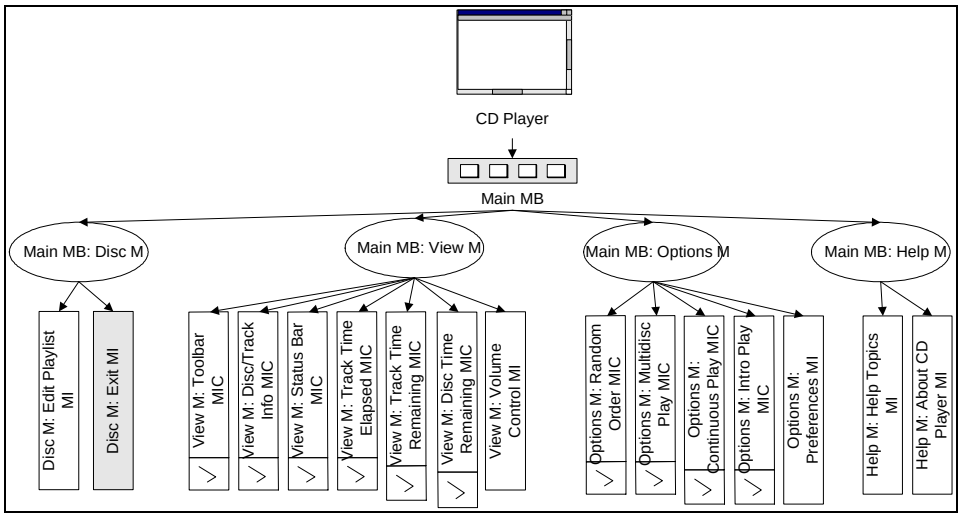


Figure 9 – The CD Player menu bar and menus.

In Figure 9, the CD Player application contains a menu bar that contains four menus: Disc, View, Options, and Help. To facilitate traceability of content, interactors and subordinate containers always reference their parent container using the naming scheme:

```
<container name> <container abbreviation> :
             <name> <element abbreviation>
```

For example, `“Disc M: Edit Playlist MI”` means the Edit Playlist Menu Item contained in the Disc Menu. Parent containers are typically the application itself, windows, and dialog boxes.

Figure 10 shows the entire CD Player Format menu and the Font dialog box that can be accessed from it. The arrows from the menu item “Font MI” connect to the Font dialog box, showing navigation. The arrows from the Font dialog box to its morphological elements show containment. Note that the Font dialog box, as a top-level container, does not reference any parent.

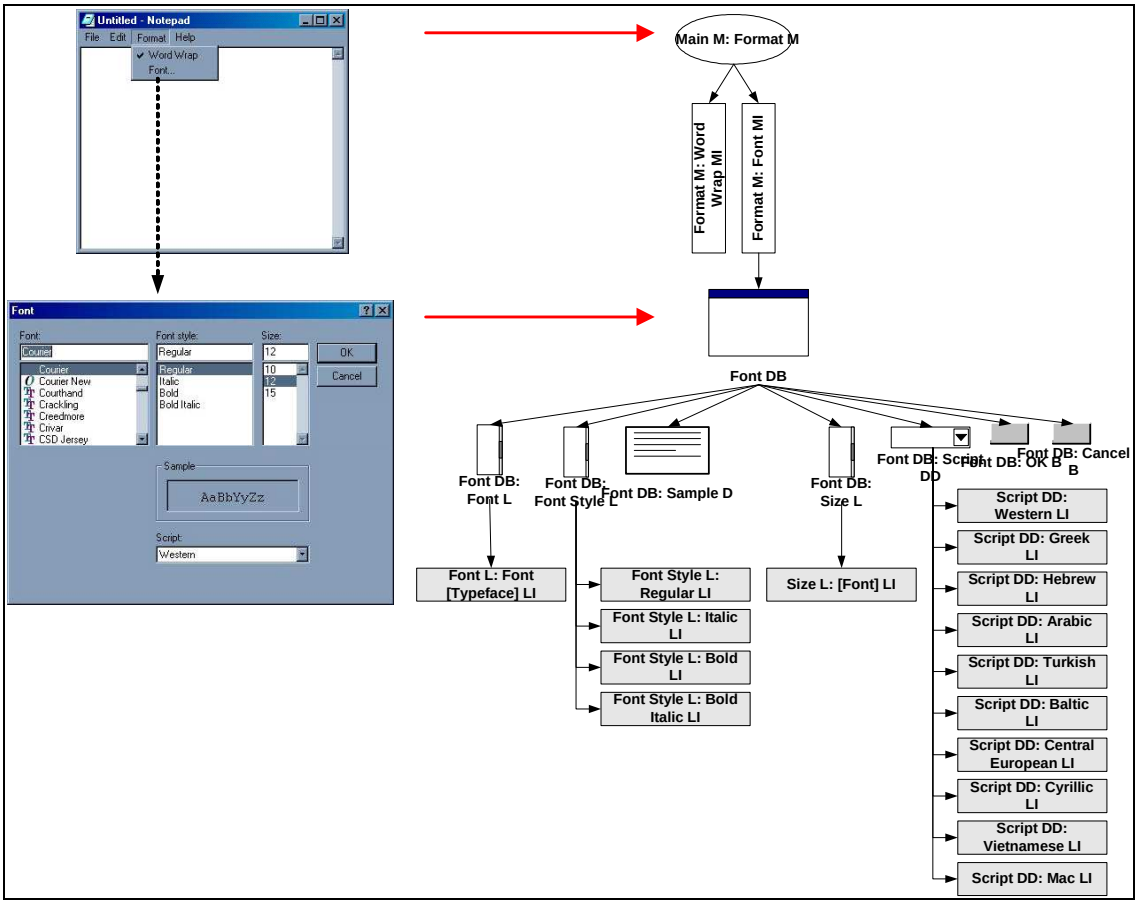


Figure 10 – The CD Player Format Menu and Font Dialog Box

3.7 Inferred and Specified Names

Whenever possible, we specify a textual identifier for the element using the corresponding label from the interface. Occasionally, for the sake of specificity or to generalize a list, we create a term to supplement an element name. We enclose such terms in brackets to distinguish them from the morphology labels. An example of this model-specified labeling in Figure 10 is the “Size” item in the Font dialog box. In Notepad, fonts have a size. However, using the morphology’s label “Size” can result in confusion if another item in the application also has a “Size”. To avoid this problem, we add our term “Font”, and put it in brackets to show that we applied our naming scheme to the label found in the dialog box. The resulting name,“[Font] Size”, can be easily found in a list of morphological elements.

3.8 Modes of Operation

Some features require the application to shift into a different mode of operation, displaying a different overall morphology than its normal state. We model this state change by enumerating the alterations to the displays and morphological elements that shift modes, as well as connecting these changes to the morphological element that produced the change of mode.

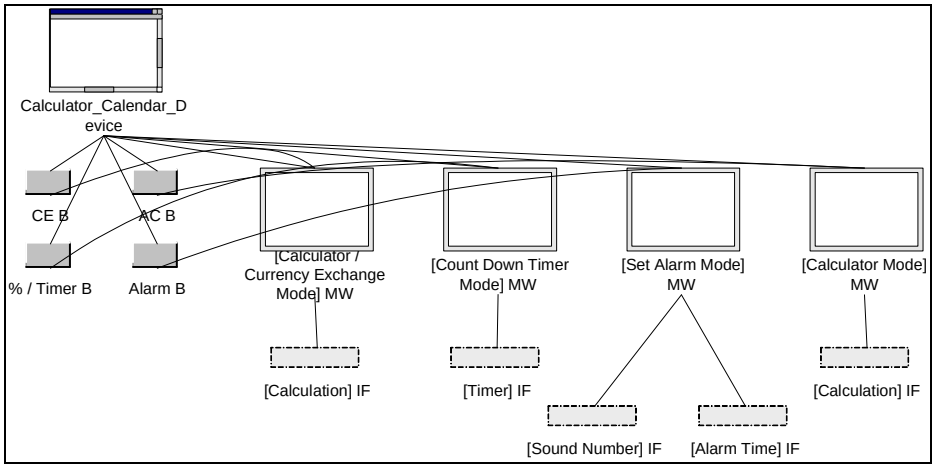


Figure 11 – Portion of Calculator / Calendar Morphology Showing Modes of Operation

Figure 11 shows part of the Calculator / Calendar Morphology. The device has one main display that switches modes depending on whether the user is accessing the currency exchange calculator, setting and using the count-down timer, setting the alarm, or performing normal calculations. We map this situation by showing the application with a main window (MW) that has four different modes of operation. Each mode has its own morphological elements. We also show the four buttons that activate these modes of operation and trace this behavior using a connection arrow. The application also has a primary display that always shows the time and calendar (not shown in the figure).

In cases where an element like a dialog box drives the modal behavior but no other changes appear in the display, we simply model that element and do not note that a change of mode has occurred. For example, in some applications, the Find / Replace dialog box causes the application to shift into a mode where the user can only search for text but not perform any other operations until they exit from the dialog box. In these cases, we only model the Find / Replace dialog box and its components.

3.9 Establishing Application Boundaries

Desktop applications often access functionality from the operating system, the Internet, or other applications. Surveying morphologies beyond the boundaries of the computing application being studied will produce an inaccurate model of the ontology, which will contain numerous concepts that do not technically belong to that application.

3.9.1 System Elements

Morphological elements that can be identified as specific to the computer or operating system running the application are not modeled, partly for simplicity, but mostly because the concepts these elements express belong to the system, not the application. These include keys on a keyboard, mouse movements, or shared printing capabilities. For generic functionality common to all applications on a system, we create a specific morphological element that can be assumed to have the basic components associated with those functions. For example, one convention we created is a special type of dialog box that we called a “File Handling Dialog Box”, abbreviated “FHDB”, to use whenever an application’s morphology had elements that opened or saved files. The FHDB assumes that the dialog box contains elements such as directory buttons, a directory listing, and configuration settings for displaying the directory.

We also exclude behaviors that are controlled at the system level. If the application does not directly control or change attributes of the element in question, we assume it to be a system function. For example, lists of fonts (or font typefaces) are managed by the operating system and management of these lists takes place outside the application. In Figure 10, we simply model this by showing one generic “Font [Typeface]” list item connected to the “Font” list. We could have chosen to model the entire list of font

typefaces as list items. However, the individual typefaces have nothing to do with the ontology. One user could have 700 typefaces and another could have 10 typefaces. Neither user would notice any differences in Notepad's functionality because the operating system manages font typefaces. However, if the computing application being studied allowed a user to edit and modify font typefaces, then the list of available typefaces that came with the application should be modeled in a morphological. As another example, the CD Player does have a volume command, but it activates the volume control dialog box of the operating system. Thus, we do not model the volume control in the CD Player's morphology. Naturally, if we were excavating the ontology of the operating system itself, we would model this shared functionality.

3.9.2 Supporting Applications and Embedded Objects

Ideally, we would like to model only those things within the scope of what we consider to be the morphology of the application being studied. However, making this distinction becomes difficult when applications access items such as embedded objects created by supporting applications. For example, in Microsoft Word 2000, a user can insert a spreadsheet from Microsoft Excel into the Word document and use Excel's spreadsheet operations. However, this changes the user interface of the application to reflect Excel functionality. To resolve ambiguities about where an application boundary falls, we adopt the following rule: If a morphological element, usually a container, can be clearly identified as being a separate application, we do not model it. For example, external applications often have their own menu bars and toolbars, making them easy candidates for exclusion. Some grey areas include items like Wizards, which acts as an application that provides guided assistance to users to perform specific tasks, and shared

functionality like graphics tools in the Microsoft Office suite. When we cannot determine that an element is really external to the application being studied, we model its morphology.

3.10 Abstraction of Interface Structure

Because the morphological map to guide our excavation of the underlying concepts, we did not design morphological cartography to model the user interface with absolute fidelity to its visual appearance. To simplify our representations, we abstractly denote certain interface elements using simpler forms, such as lists or interactive displays. For example, Figure 12 shows the Color Palette display used by Windows applications. A user can move a mouse pointer anywhere in that palette and select a color. Rather than model 140 separate buttons, which might be reasonable given that selecting a hexagon changes the color selection, we choose to model it as an interactive display for simplicity.

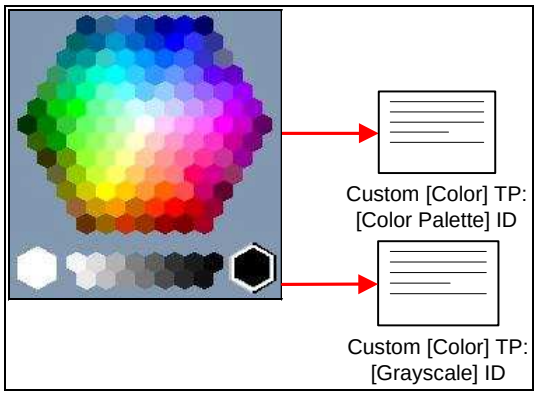


Figure 12 - Color palette represented as an Interactive Display

While this abstraction fails to capture the elegance of structure or the color arrangement by a color's relationship to wavelengths of light, for the purposes of identifying the ontology concepts, capturing the forms is not necessary.

3.11 Automation of Morphological Cartography

There have been a number of approaches for reverse-engineering user interface models for the purposes of reuse and testing. MORPH, the Model-Oriented Reengineering Process of Human-Computer Interfaces, uses static code analysis and recovers interface designs from character-oriented user interface designs and transforms them to graphical user interfaces [146] using an abstraction hierarchy based on Foley's basic interaction techniques [66]. CelLEST reverse engineers legacy interfaces based on user interactions and the construction of a GUI or a web interface from these interactions [182, 183]. GUITAR is a system for testing GUIs that automatically generates a GUI representation through interaction with the application. The GUI is expressed as a graph with nodes denoting window states. Windows are modeled in terms of the set of widgets (e.g. buttons, labels, text fields) that comprise the window, its properties, and the values associated with those properties. It also distinguishes between modal and modeless windows. However, it does require human intervention, as GUITAR occasionally will stop modeling at some dialog boxes [143].

Our manual process for mapping the morphology of an application can be time consuming and prone to naming errors. However, this process more than compensates for these shortcomings by allowing preciseness of modeling application behavior and flexibility of usage – we can apply these methods and representations to any computing application, device, embedded system, or any information artifact with an accessible

morphology and underlying ontology (e.g. a web site). A future tool for supporting developers will incorporate some of the ideas and technologies from previous research in the area but will implement features that facilitate the design, reverse-engineering, and semi-automated generation of an application morphology. The process of reverse-engineering an application morphology will always be semi-automated. However, processes such as automatic generation of the map and tracking the labels used would reduce the effort required and reduce errors in morphological coverage.

4 ONTOLOGICAL CONSTRUCTION: MODELING THE ONTOLOGY

In this chapter, we present our methods for excavating and constructing an application's ontology from its morphology. First, we systematically identify concepts from the list of elements generated from a morphological map. Then, through a combination of static and dynamic interactions with the application, we identify the relationships between these concepts. Finally, we model the application ontology as a semantic network composed of these excavated concepts and relationships.

4.1 Ontologies in Computing

Traditionally, the word "ontology" refers to "a branch of philosophy dealing with the *a priori* nature of reality" [36, 37, 79]. In computer science, *ontology* is used to describe a set of concepts or representation of these concepts for domain and data modeling. However, the grounding provided by the philosophical formalisms has been used to refine and concretize data modeling formalisms [79-81, 193]. Ontologies in computer science have been designed for representing knowledge in intelligent systems [19] and for exchanging data between knowledge databases used in applications such as web searches and e-commerce [30, 32, 77, 78, 126, 141, 142].

Some ontologies have text-based ontological notations designed to support data modeling and database exchange activities, such as Ontolingua [77], CLASSIC [32], and CYC [126]. Other representations exist for modeling application concepts in diagrams, such as entity-relationship (ER) diagrams [15, 23, 45, 73], object-role models (ORM)[85, 151, 191], and object-oriented (OO) diagrams [25, 26, 62, 175]. We have chosen to

model our recovered ontologies diagrammatically – as a connected graph, or *semantic network*, of concepts and relationships [16, 152, 177, 180].

4.2 Overview of Ontological Construction

The basic steps of morphological cartography are:

1. Excavate Concepts – Identify nouns and objects from the labels of morphological elements.
2. Identify Relationships – For each concept, identify the relationships it has with other concepts in the ontology by examining the user interface and by interacting with the application.
3. Assemble the Semantic Network – Connect the concepts together with their relationships.

4.3 Step 1: Excavate Concepts

Abstractly speaking, a *concept* is a generalized idea of a thing or class of things [177]. In our excavated ontologies, a concept is any thing that has a name or some concrete implementation in the computing application of interest. We organize these concepts into three different data modeling categories: entity types, attributes, and instances.

- An *entity* is a named thing that can be distinctly identified [45]. A set of entities that share a set of attributes is an *entity type* [59]. Example: In Figure 16, Disc and Track are entity types.
- An *attribute* is an intrinsic property of a thing in the real world [192]. An attribute is a concept lacking independent existence except as a property of an entity type. Example: In Figure 16, Track Name and Track Number are attributes of Track.
- An *instance* is a concrete manifestation of an entity type [26]. We do not model instances in our ontologies. Rather, we use instances to identify entity types. For example, “Times New Roman” is an instance of the “Font [Typeface]” entity type.

4.3.1 Identifying Concepts

Using the morphological map as an information source, we identify the concepts from the nouns and inferred nouns in the labels attached to morphological elements. This technique of looking for noun phrases and the indirect objects implied by verbs originates from data modeling and object-oriented analysis methods [25, 175]. For example, a “File Menu” implies that there is a concept of “File”; a “Font” dialog box contains the concept “Font Size”. In desktop application morphologies, we ignore references to morphological elements, such as windows or toolbars.

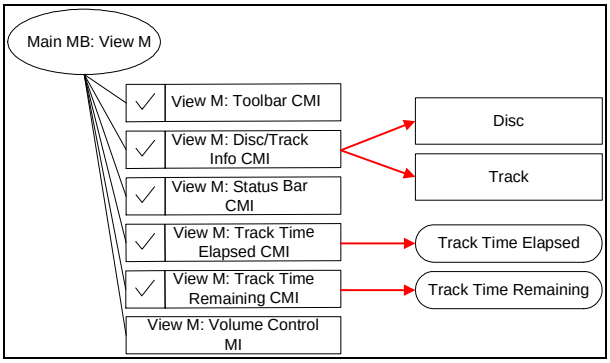


Figure 13 – CD Player View Menu – identified concepts and attributes

Figure 13 shows an example of concept identification from the View menu of the CD Player. The menu allows the display of the CD Player toolbar, Disc and Track info, the Status Bar, the Elapsed Track Time, and the Remaining Track Time. It also allows the user to change the volume by accessing the operating system’s volume control application. We ignore the morphological containers, Toolbar and Status Bar. We also ignore Volume Control as a system function. Disc and Track are nouns and we model those as entity types. Track Time Elapsed and Track Time Remaining do not have any

meaning except in the context of a track being played. Because they lack an independent identity, we model them as attributes.

4.3.2 Inferred Names

When the label for a morphological element fails to specify a concept adequately or completely, we enhance it with our own terminology. Like the inferred names in the morphology, we enclose these terms in brackets to indicate the use of language not found in the morphology. In Figure 10, we used the term “Typeface” and enclosed it in brackets next to “Font”. The word “Font” in desktop applications often refers to both the general font setting of text and what is correctly known as a “Typeface”, which is a set of character tokens with a specific appearance. “Courier, “Times Roman”, and “Helvetica” are examples of typefaces and so the list in the Font dialog box is really a list of font typefaces. We named the generic list item in the list a “Font [Typeface]” to distinguish it from the general term “Font” in Notepad.

4.4 Step 2: Identify Relationships

After identifying the concepts from the morphology, we identify the relationships between them by interacting with the system and reconstructing them from observations of both static information and dynamic behavior. For constructing a semantic network, we use basic relationships from object modeling: generalizations (*is-a*), aggregations (*has-a*), and associations [26].

4.4.1 Generalizations

A generalization is a relationship between a kind of thing (the *parent* or *superclass*) and a more specific kind of that thing (a *type, child,* or *subclass*). A *subtype* is a

specialization of an entity type. In object-oriented methods, the generalization relationship also means that a child inherits the attributes of the parent. We use this relationship (and generate the corresponding entity types and attributes) when we identify several objects with the same set of attributes. We model a generalization using an arrow labeled with “is-a” originating from the entity type to its parent.

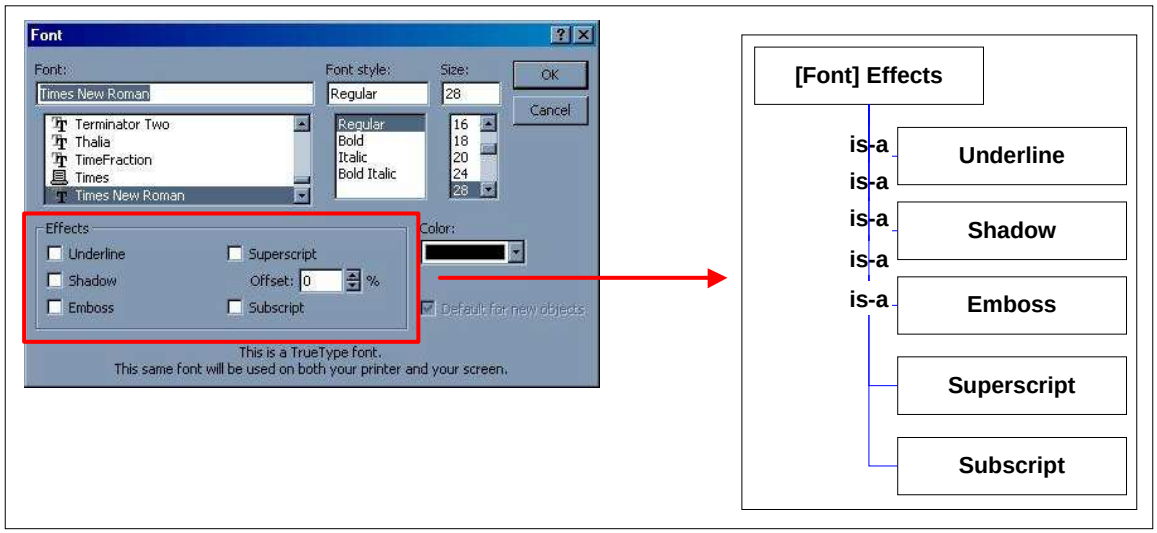


Figure 14 – An example of a generalization

Generalizations can also be identified from morphological containers that contain lists of uniquely named items under a common heading. In Figure 14, we show a Font dialog box for formatting fonts. There are five different checkboxes within the space labeled “Effects”. From this we identify the concept “[Font] Effect” and five types of effects. From this dialog box we can also identify the concept “Font Style” and the four different types of font styles available.

Occasionally, we create a subtype from an existing entity type if we detect a set of instances with a slightly different set of inherited properties or if, for modeling correctness, we need to account for operations on the entity type.

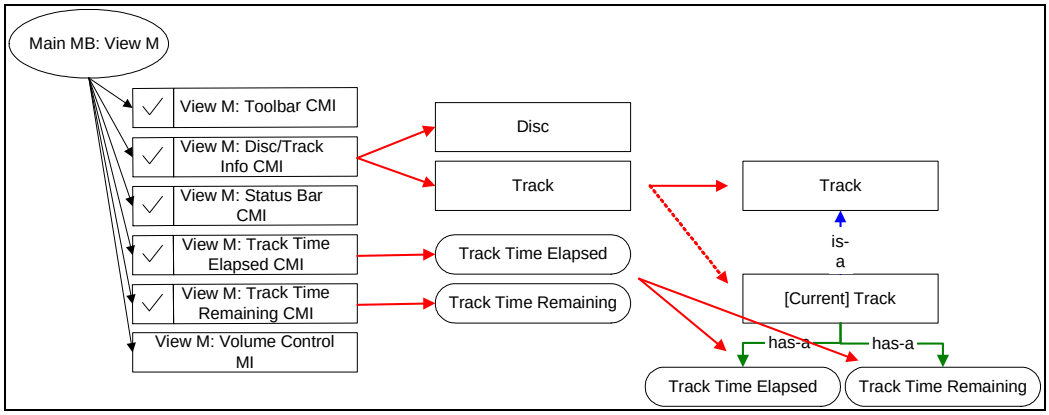


Figure 15 – Example of an inferred concept and relationship identification

As shown in Figure 15, we created the entity type "[Current] Track" as a type of Track for our CD Player ontology. Using the morphological element labels, we identify the concepts "Disc" and "Track". We created "[Current] Track" to represent the track currently being played by the user. We make this distinction because the tracks on a CD cannot simultaneously possess a Time Elapsed or a Time Remaining. These attributes can only belong to a currently playing track. However, [Current] Track still possesses the basic attributes of a track, so we model a generalization relationship between [Current] Track and Track.

4.4.2 Aggregations

An aggregation is a whole/part relationship used when an entity type is composed of other concepts – attributes and other entity types. We model an aggregation relationship with a 'has-a' arrow in the semantic network.

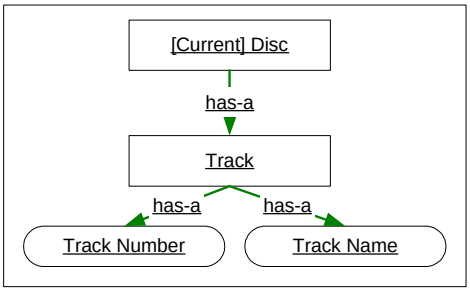


Figure 16 – An example of an aggregation

For example, in Figure 16, we show examples of aggregations. A [Current] Disc is composed of Tracks. A Track has the attributes Track Number and Track Name. Aggregations can be identified from dialog box settings for specific items or from a list of properties associated with the concept in question.

4.4.3 Associations

We use associations to indicate relationships between concepts not covered by generalizations or aggregations. An association is a structural relationship that specifies that things of one type interact with things (concepts) of another type.

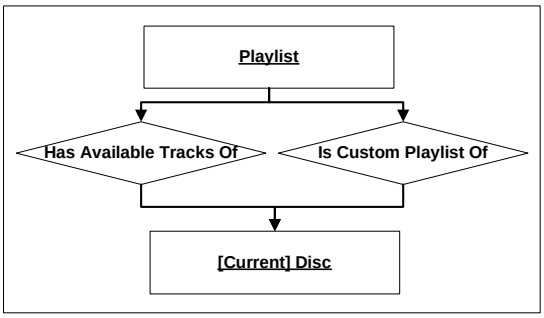


Figure 17 – An example of an association

In the CD Player application, a playlist can be modified to play a customized sequence of tracks available on the current disc playing, and it can display the available

tracks in their original sequence. We show this in Figure 17 as two associations between [Current] Disc and Playlist. Following the arrows, this relationship can be read as “A Playlist has Available Tracks of [Current] Disc” and “A Playlist Is Custom Playlist Of [Current] Disc.”

4.4.4 Dynamic Identification of Relationships

Many relationships can be identified through static examination of the morphological elements. For example, dialog boxes allow the user to modify properties of a specific item. Labels and text in those types of morphological containers often provide hints about that items attributes and the other objects that interact with the item. However, computing applications often possess complex relationships that only become evident when the application is performing an operation or by observing changes of state after an operation was performed. Because ontological excavation is performed manually, the attention to this level of detail in relationship identification can affect modeling precision.

For example, in Notepad, a user can change the font and the script in the Format Menu and this changes the global appearance of the letters in the Main Window. However, this actually has no effect on the file that Notepad opens and saves. In fact, Notepad will apply those font settings to the next file that is opened. We identified this behavior during the course of determining what properties were saved with the file. We created a file, modified the font settings, then opened a different file and saw it displayed with the same settings as the first file. We chose to use the concepts “[Configuration]”, “Document”, and “[Current] File” to explain this behavior. Notepad opens a File which becomes the [Current] File. The [Current] File is considered to be the Document displayed by Notepad and can be printed with Headers and Footers or displayed with

settings such as Font and Word Wrap. In short, the [Current] File stores the edited Text; the Document displays it.

4.5 Step 3: Assemble the Semantic Network

The basic structure of a network, semantic or otherwise, consists of nodes and edges, allowing us to use graph algorithms to identify key elements in the ontology. Also, this simple representation can be enhanced and refined into any of the aforementioned software design notations. To assemble our ontological model using the semantic network form, we simply model concepts as nodes in a graph connected by edges representing their relationships, as we have shown in several of our previous examples.

Figure 18 shows the completed ontology for the CD Player. A complete list of our symbols and abbreviations can be found in Appendix D.

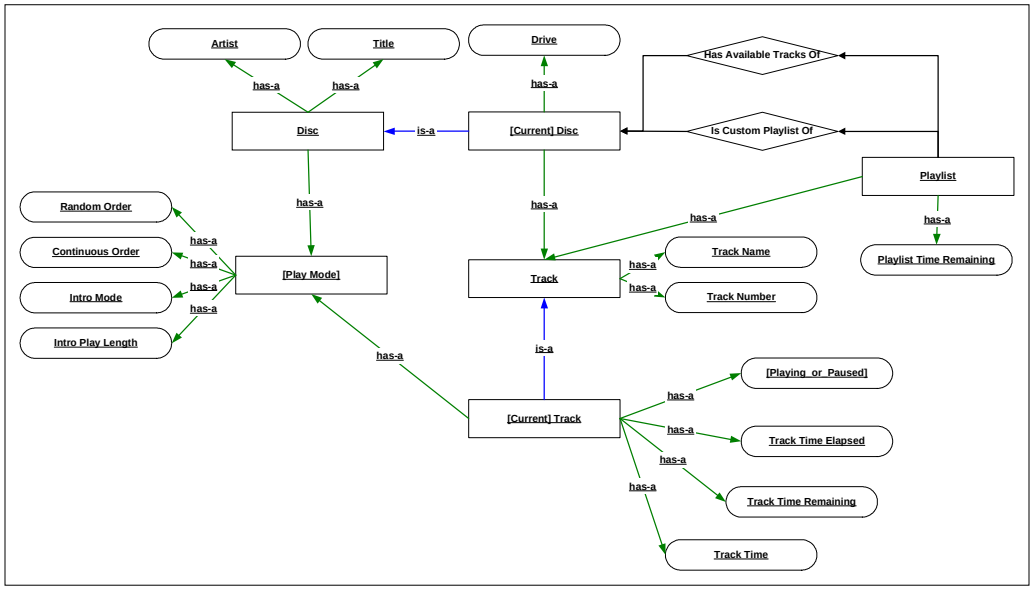


Figure 18 – The CD Player Ontology

4.6 Modeling Attributes as Nodes

In object-oriented models, such as those using the Unified Modeling Language (UML), attributes of classes are modeled within the structure used to describe the class [25, 26]. This convention allows developers to ensure that proper information hiding and modular design of the objects. Figure 19 shows a UML model of the CD Player. It has six classes that correspond to the six entity types in our semantic network (Figure 18) and no other features except for the edges that connect the classes. This diagram, generated with a Microsoft Visio template, also contains information about the cardinality of the relationships between the classes.

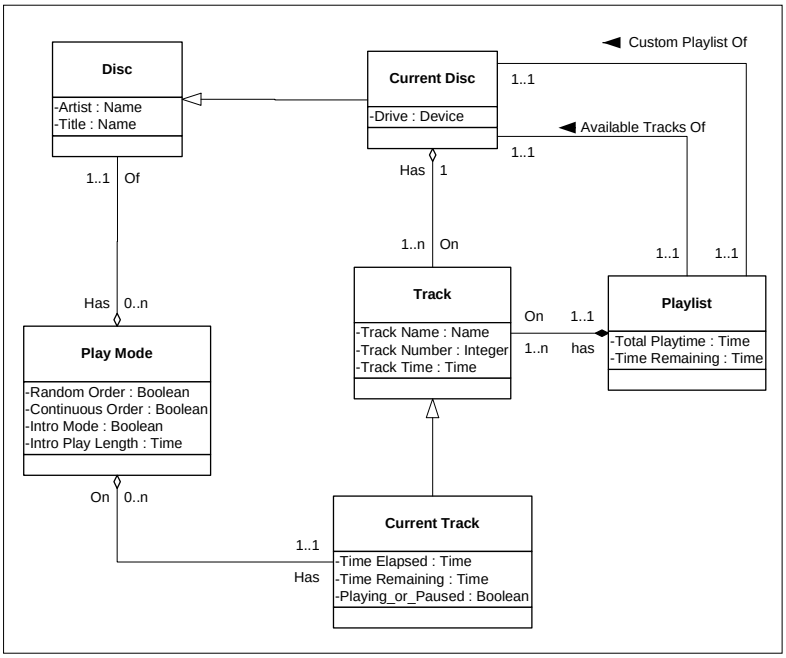


Figure 19 – UML model of the CD Player

In object-oriented methods, enforcing information hiding facilitates modeling the objects as encapsulated units that contain behavior and state. For our work, we want to expose the complexity and potential interactions between a concept's attributes and

potential interactions within the ontology. Thus, to leverage the structure of the semantic network to our best advantage, we find modeling the attributes as nodes enriches the graph, allowing us to apply the graph metrics and analysis techniques that we discuss in Chapter 5. This convention is also used in NIAM (Natural language Information Analysis Method) [191] and ORM (Object Role Modeling) [84, 151].

4.7 Specific Modeling Conventions

4.7.1 Options and Tools

Desktop applications have settings and options that allow a user to configure the behavior of the system and its specific features. To model global system settings that affect the general behavior of the application, we create an inferred entity type "[Configuration]" that contains all the rules and settings. We model these rules and settings as attributes of the entity type whose behavior they modify, using the text "Rule:" For example, a checkbox morphological element labeled with the text "Show Highlight" becomes a part of the application's "[Configuration]" as "Rule: Show Highlight".

We also created a naming convention, "[Tool]", to account for features that perform tasks in the application. For example, the feature that searches through a document for a piece of text becomes a "Find [Tool]" in our ontology. Often these features are really smaller applications subordinate to the application of interest. We use this convention as a placeholder to capture the various attributes and rules expressed by that smaller application.

4.7.2 Unique Naming

As a rule, all concepts should have unique names. While generally, concepts should adopt the names as used by the morphological elements that express them, applications often use labels contextually. The word “Size” or “Category” may appear next to several different concepts. When a duplicate name is found, it should be enhanced with additional information to make it more specific. For example, “Font” is often used interchangeably in a dialog box to mean both the format setting and the typeface. Thus, we label the concepts “Font [Format]” and “Font [Typeface]” to prevent confusion.

4.7.3 Primitive Data Types

Most modeling conventions for building computing applications contain rules for specifying primitive data types, such as strings, integers, and real numbers. We do not model these programming data types in our ontologies. While types are important to understanding the implementation of a program, they are not necessarily important to understanding a problem domain. For example, in PowerPoint 2000, a Slide has a Slide Number which is an integer. We do not model the Slide Number as an integer in the ontology.

4.7.4 Constraints

Other software development notations show constraints on relationships such as cardinality (the number of elements in a set) or dependencies (a semantic relationship between two things) [15, 25]. These conventions typically appear on the edges near the object with the constraint. We chose not to model them because they were not essential to our analysis and did not contribute any structure to the semantic network. However, the

abstraction of these ontology models allow them to be easily enhanced with constraints if necessary – for example, to generate software architectures for product development.

4.8 Establishing the Boundary of the Ontology

Excavating concepts into ontologies can threaten to become a never-ending exercise if the modeler attempts to define all interacting applications or to define concepts as deeply as possible using information not directly expressed by the morphology.

4.8.1 Application Boundaries

Many applications on desktop systems access functionality using shared operating system modules, accessing the Internet through external web pages, or partnering with other applications through embedded components. Productivity packages, such as Microsoft Office, use links and embedded objects to share features across the individual applications. Thus, establishing the boundary of the ontology of interest is necessary to keep the models tractable and to ensure that the analysis returns the sound conclusions. Once a computing application of interest has been identified, we recommend using the heuristic of excluding the following from that application's ontology:

- Other applications. For applications on personal computers, we chose to ignore those external windows that possessed their own menus and toolbars, treating them as external applications. Naturally, what defines a separate application is ultimately up to the modeler.
- Web pages that do not change any state of the application being studied. In instant messenging applications, actions such as changing a password or certain window settings invoke an external web page from the distributor's servers. In turn, those web pages invoke other web pages. We included web pages that modified the application being studied and excluded those web pages that had no visible effect on the application. For example, for Yahoo! Messenger, we model the web pages

that allow a user to change the stocks listed in a portfolio but ignore those pages that display the performance of a company.

4.8.2 Conceptual Boundaries

Another type of boundary concerns the depth of various concepts. We use a very simple set of conventions in identifying concepts and relationships. If something has a name in the application and it has visible behaviors, we model it. However, we do not extend the model beyond the named concepts in the morphology. For example, we did include the Calculator / Calendar's mathematical operators such as multiplication and division. We did not model the underlying ontology of basic mathematics that define these operations (e.g. that multiplication, *a* * *b*, is the process of adding some quantity *a* to itself *b* times). In Notepad, we did acknowledge that there were four different types of Font Types. We did not take extra effort to explain how those work or why they exist.

Concepts that have been operationalized in an application often have real world meanings that are assumed to be common knowledge on the part of their users. For example, by digging a little deeper and by looking at the rules implemented, one could recover an ontology of English grammar from a grammar checker. However, this depth would have to be consistently applied for all other concepts identified in the application. The Calculator / Calendar includes sixteen different time zones, identified by the names of major cities in the world for that particular time zone. However, nothing else in the morphology reveals anything meaningful about what those strings of characters mean. "New York" has a time zone three hours behind "Los Angeles". From the standpoint of the application, the labels on the keys could easily be "blue" and "aardvark" while still expressing the same behavior. We do model the concepts "New York" and "Los

Angeles" as types of "Time Zones" – where "Time Zone" is obtained from the instruction manual accompanying the device. However, we do not try to introduce second-order concepts beyond what can be identified from the morphology.

4.9 Automation of Ontological Construction

Understanding the domain descriptions from either a computing application or from work products developed through user interactions are an important prerequisite to building (or reverse-engineering) a useful computing application. Previous work in domain analysis and reverse engineering has developed methods for extracting the domain from program documentation [3], requirements specifications [74], code [47, 56], and interviews with domain experts [8]. Of these techniques, code domain analysis might be the best method for automating ontological excavation but code itself contains a meta-domain with concepts and relationships that concern software engineering and programming. If we wish to preserve the black-box nature of ontological construction, then building tools to aid the development of a model would certainly reduce the effort required, but we believe that full automation would be extremely difficult.

5 ONTOLOGICAL ANALYSIS

In Chapter 1, we claimed that all ontologies have core concepts that could be identified and that concepts either contributed to or detracted from the conceptual coherence of an application. We have explained and demonstrated how we excavate ontologies from computing applications. We now present our methods for analyzing them. Because we modeled ontologies as semantic networks, we can analyze them using techniques from graph theory to accomplish the following:

- Identify the core concepts of an ontology
- Identify tightly connected groups of concepts that we call *teleons*
- Measure the conceptual coherence of an ontology

In this chapter, we present our methods for ontological analysis along with examples from case studies that demonstrate their usage.

5.1 Graph Theory and Ontological Analysis

Our semantic network is a graph – a mathematical structure consisting of nodes that are connected by edges. Graphs are well understood structures in both mathematics [86, 87] and computer science [48, 176]. They are a natural means for visually and mathematically representing how a set of items are related. They have been applied in architecture to derive the transit patterns in cities and the living patterns of inhabitants in buildings [90, 91]; in software engineering to model application states [88], data flow [167], and conceptual relationships [45]; in database methodologies to model data structures [20, 59]; and in artificial intelligence to model knowledge [16, 127].

To analyze our ontologies, we adopted graph metrics and analysis techniques used in social network theory. Social network theory uses graphs to model the interactions among social entities, which are individuals and organizations. Theorists can use these graphs to determine, for example, which social entities have the most influence in the network, what information flows exist in the network, and whether specialized subgroups exist within the network. They accomplish this by applying metrics and algorithms from graph theory to identify both patterns and variables in the structural relationships of these networks [194].

5.2 Generating Graphs from Ontological Diagrams

We model our excavated ontologies in a Microsoft Visio diagram, using a template and stencil that we created specifically for this work. We wrote a Visual Basic macro that transforms the diagram into an undirected graph represented as an adjacency matrix (see Appendix C).

In graph theory, an arrowed line denotes a directed edge. In a directed graph, node A can only reach node B if a directed edge or a path of directed edges exists, such that it originates from A and terminates at B. In our ontological diagrams, we use arrowed lines between the concepts to represent relationships. However, these arrows serve only to support readability, not to represent the semantic network as a directed graph. We tested our methods on both directed and undirected graphs and discovered that modeling ontologies as directed graphs produces too many isolated components that confound the analysis methods. Hence, we decided to use undirected graphs for our analysis.

We use a social network analysis tool called UCINET [29] to analyze the graphs generated from our ontology models. This tool implements numerous graph algorithms and metrics specifically chosen to study attributes of social networks.

5.3 Identifying Core Concepts

5.3.1 The Role of Core Concepts in an Ontology

Core concepts form the conceptual foundation for an ontology. Without these core concepts, the ontology could not exist in its current form. Other concepts in an ontology either exist to support and define core concepts or are not required in the ontology and reduce its overall conceptual coherence. Because of their role in defining the framework of the ontology, core concepts relate to many other attributes and concepts. When modeled using our method, these concepts appear as nodes found in the spatial centers and foci of a graph, often connected by edges to other centrally-placed nodes. Thus, core concepts serve as critical nodes that provide the structure of an ontology.

5.3.2 Prestige Measures in Social Network Theory

Social network theorists use graph measurements that they call *prestige measures*. Prestige measures assess the importance of a node relative to the rest of a graph. For example, *degree centrality* counts the number of edges incident to a node (or in- and out-degrees in a directed graph). A person with a high degree centrality in a social network is important because they possess many direct connections to other individuals. Prestige measures have been used to study relationships, such as the Medici family and their marriages to other families during the Italian Renaissance [194]. In a previous study [96],

we applied the following centrality measures to ontologies excavated from our case study applications:

- *Degree Centrality* measures the number of edges incident to a node. The more edges on a node, the higher the centrality.
- *Closeness Centrality* measures the average distance from that node to all other nodes.
- *Betweenness Centrality* measures the number of shortest paths between all pairs of nodes in the graph that contain a particular node. The higher the centrality measure, the more dependencies on that node. Because leaf nodes only serve as start and end points for paths, they have a betweenness value of 0.
- *Information Centrality* measures the information contained in all paths originating with a specific node.
- *Eigenvector Centrality* measures the centrality of a node relative to the importance of its surrounding nodes.

We determined that betweenness centrality best identified the core concepts in an ontology. It not only had the lowest sensitivity to small errors in the model but returned what we considered to be the most intuitively correct core concepts in the graph. It also had the additional benefit of ignoring attributes (modeled as leaf nodes) in the analysis.

5.3.3 Betweenness Centrality

Betweenness centrality measures the number of shortest paths that include a particular node. It computes a normalized value between 0 and 100 percent where 100 means that the node lies on all shortest paths between all pairs of nodes. In the social network sense, it calculates the probability that a "communication" (i.e. a path) from a social entities *j* and *k* use this particular social entity as an intermediary. It assumes all edges have equal weight and that communications travel along a *geodesic*, the shortest

route between a pair of nodes. Where there are multiple geodesics, any can be used, so betweenness centrality assumes that all geodesics are equally likely.

Calculating the betweenness centrality of a node in a graph uses:

- g_{jk} – the number of geodesics between *j* and *k*.
- $(g_{jk})^{-1}$ - if all geodesics between *j* and *k* are equally likely to be chosen, the probability of a communication using any one of them.
- $g_{jk}(n_i)$ – the number of geodesics linking actors *j* and *k* that contain node *i*.

The probability that a distinct node *i* is "involved" in the communication between node *j* and *k* can be computed as:

- $\frac{g_{jk}(n_i)}{g_{jk}}$

Thus, the betweenness centrality for n_I expressed as a sum of estimated probabilities over all pairs of nodes can be expressed as:

- $C_B(n_i) = \sum_{j<k} \frac{g_{jk}(n_i)}{g_{jk}}$

And the scaled and standardized index of centrality can be expressed:

- $C'_B(n_i) = \frac{C_B(n_i)}{[(g-1)(g-2)/2]} * 100$

The index counts how "between" each of the nodes is as a sum of probabilities. The minimum is 0 if n_i doesn't fall on any geodesics. The maximum is (g-1)(g-2)/2, the number of pairs of nodes not including n_i [194].

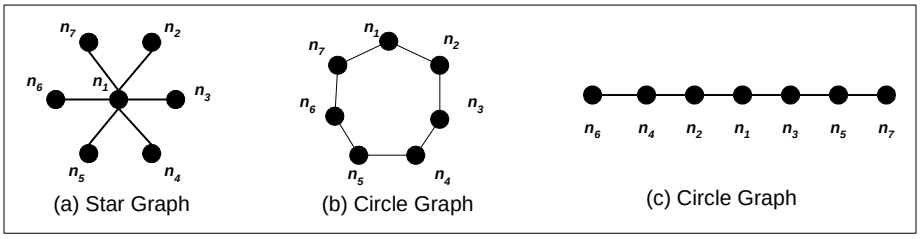


Figure 20 – Examples for betweenness centrality measures

In Figure 20, we show three simple graphs. In the star graph, $C'_B(n_1) = 100.0$. The rest of the nodes have a C'_B of 0.0. The value of 100.0 simply expresses that all paths with a length greater than 1 use the central node, n_1. In the circle graph, $C'_B = 20$ for all nodes. No node is more central than any other node. In the line graph, $C'_B(n_1) = 60.0$, $C'_B(n_{2,3}) = 53.3$, $C'_B(n_{4,5}) = 33.3$, and $C'_B(n_{6,7}) = 0.0$. The more central the node, the higher the betweenness centrality.

In future sections, we will simply refer to betweenness centrality with respect to concepts as the "centrality value" of a concept.

5.3.4 Core Concept Threshold and Sensitivity

In any arbitrary graph, nodes will have a range of centrality values. In our case studies, we determined that betweenness centrality values for concepts were not evenly distributed, but had a discontinuity at about 7.0. We used this as a heuristic for dividing concepts into two classes: core and peripheral, with 7.0 serving as the cutoff point. Concepts with values under this threshold exist to support core concepts but could be removed from the ontology. For example, in the Scheduler, "Time" has a centrality value over 7.0 while the concepts "Hour", "Minute", and "Second" fall under it. The concept of time, for the purposes of scheduling, is certainly expressed in terms of the hour, minute, and second of a day. Removing time would compromise the ontology of the scheduler.

However, any of the subordinate concepts could be removed, changing how time is expressed in the scheduler but not affecting the presence of time in the scheduler. We believe that our arbitrary threshold may be related to fundamental properties of the graph, such as the distribution of centrality values across the nodes that compose it, but do not claim any significance to that value. Although this core/periphery threshold was useful for the current investigation, future empirical studies are needed to evaluate its reliability across other applications.

5.3.5 Case Studies: Core Concept Identification

In our case studies, we identified the following core concepts from these applications:

Table 1 – Core Concepts found in the case studies. Concepts are listed in order of their betweenness centrality values

Application	Core Concepts
CD Player	[Current Track], [Play Mode], Track, Disc, [Current Disc], Playlist
Scheduler	Event, Date, To Do Item, Hot Synch, Day, Month, Time, Alarm, Repetition, Note, Every
Notepad	Page Setup, Font [Setting], Paper, Text, Paper Size, Font, Script, Header, Footer, [Configuration], [Header/Footer Code], Margins, Alignment, Font Style
Calculator / Calendar	[Time Zone], Time, Home Time

Intuitively, from our knowledge of these problem domains (e.g. CD playing, scheduling, etc.), we know that the concepts obtained by our analysis are indeed core concepts. However, our metrics only identified two core concepts in the Calculator / Calendar, neither of which concern mathematical calculation. We observed that the Calculator / Calendar ontology had isolated subgraphs in its ontology. Isolated components in an ontology make all concepts less prominent, explaining why we only

found two core concepts. We performed a separate analysis on each subgraph and obtained the following core concepts within them:

Table 2 – Core concepts found in Calculator / Calendar. Note: Subgraph 4 only has 2 nodes.

Subgraph	Core Concepts
1	Date, Month, Year, Calendar
2	[Time Zone], Time, Home Time, [Time Display Mode], Alarm Time, Alarm
3	[Mathematical Operation]
4	Currency Exchange [Calculator], Exchange Rate *

We believe that our use of betweenness centrality to the excavated ontologies identified concepts that could be argued to be core concepts in their respective applications. The exception is the Calculator / Calendar. In the case of the CD player, we identified concepts that belong to all desktop CD player applications. We expect this because the CD player's features are organized around a basic task – playing CDs. Although the Calculator / Calendar has many different features, we only identified two core concepts when analyzing the whole ontology. However, when we analyzed the independent subgraphs composing the ontology, we found the other core concepts that we expect to find in this multiple purpose application. So, applications with disconnected components composing their ontologies must require a separate analysis to identify the core concepts.

5.4 Teleon Identification

5.4.1 Teleons: Coherent Subgraphs in Ontologies

From our studies of applications, we have found that certain concepts have strong connections to one or more other concepts, such that any one of them depends on the

others to exist. For example, Tables in office productivity software have a set of conjoined concepts that define them – Column, Row, and Cell. A Table cannot exist without rows and columns, and a cell is defined by the intersection of a row and column. These concepts can be isolated as a subgraph in an ontology that we call a *teleon* after the Greek word *teleos*, which means goal [95]. These functional subgraphs can be identified through analytical means.

There are numerous types of coherent subgraphs in graph theory that can be identified in a graph. These include *cliques* (a maximal, complete subgraph of three or more nodes) and *n-cliques* (a maximal subgraph in which the largest geodesic distance between any pair of nodes is no greater than n) [48, 176]. Many of these have restrictive criteria for subgroup membership such that they could only identify structures like triples – a 1-clique consisting of 3 nodes, or smaller variations of a large tightly connected subgroup. For example, a clique analysis performed on Notepad returns five different cliques that are permutations of Alignment, [Header/Footer Code], and Left / Right / and Center Alignment Codes. We decided to use a less restrictive subgraph structure, called a *k*-core to identify teleons in an ontology [194].

A *k*-core is a maximal, induced subgraph such that each node in the subgraph has edges connecting it to *k* or more other nodes. Wasserman and Faust [194] describe it as useful for exposing regions of the graph that likely contain other subgraphs such as cliques. This method belong to a class of techniques called *data clustering* that are used to identify significant subgroups in a dataset or graph [83, 106]. Besides applications to social network theory [28], these techniques have been applied to information recognition

[106], Web topologies [98], and the reverse engineering of objects from legacy code [189].

5.4.2 Case Studies: Teleon Identification

When we applied our *k*-core analysis to our case study applications, we identified the following subgroups (Table 3, Table 4, Table 5, and Table 6):

Table 3 – CD Player Teleons Identified by *K*-Core Analysis

***k*-value**	**Concepts in Teleon**
2	[Current Track], [Play Mode], Track, Disc, [Current Disc], Playlist

Table 4 – Scheduler Teleons Identified by *K*-Core Analysis

***k*-value**	**Concepts in Teleon**
2	PurgeUnits, Week, Month, Every, Today, Day, Preferences, End Date, Repetition, Schedule, Year, Date, Due Date, All Occurrences, Current Occurrences, Event, To Do Item, Is Private, Start Time, End Time, Alarm, Alarm Units, Hour, Minute, Backup Copy, Note, Event Problem, Synch Problem, To Do Problem, To Do List, Hot Synch, Time, Application

Table 5 – Notepad Teleons Identified by *K*-Core Analysis

***k*-value**	**Concepts in Teleon**
3	Text, Header, Footer, File Name, Page, Number, Date, Time
2 (a)	Header/Footer Code, Left/Right/Center Alignment, Alignment (of Header/Footer)
2 (b)	File, Current File, [Configuration], Line, Word, Font [Setting], Page Setup, Document, Page

Table 6 – Calendar / Calculator Teleons Identified by *K*-Core Analysis

***k*-value**	**Concepts in Teleon**
2	Date, Month, Year, Calendar
2	Alarm, Alarm Time, Count Down Timer, Hour, Minute, Second, Sound, Time,

The CD player's 2-core (Table 3) consists of those concepts which are entity types in the ontology. The Scheduler's ontology (Table 4) has such a large number of concepts in its 2-core to the point where it is nearly indistinguishable from the ontology itself. The Notepad (Table 5) and the Calendar / Calculator (Table 6) ontologies produced interesting results. The Notepad ontology had one large cluster which could be broken into 3 distinct *k*-cores. The concepts in the 3-Core all concern Text, which we expected would be the case since MS Notepad is a text editor. The concepts in the 2-cores involve Header/Footer Codes and File Handling / Document Format, respectively. Thus, the *k*-core technique shows that MS Notepad has Text, Header / Footer Management, and File Handling and Document Formatting features. The Calendar / Calculator device had two 2-cores related to the calendar date and timekeeping. However, it did not show that the device also had a countdown timer, a calculator and a currency exchange calculator.

While *k*-cores does not reveal teleons at the granularity that could directly identify features at the morphological level, it does show interesting relationships amongst the concepts in these applications.

5.5 Measuring Conceptual Coherence

5.5.1 Conceptual Coherence and Average Distance

Conceptual coherence is the extent to which the concepts in the ontology are tightly related. This notion of coherence can be found in studies of prototypical instances where subjects were given a list of items and asked to weight their resemblance to a category. In a typical study of the organization of human semantic memory, subjects were asked to consider Robin, Eagle, Ostrich, and Penguin in the category of Bird. In this example, birds that were large or flightless were considered less like birds than those that were

small or could fly [177]. Biologically speaking, all of these are birds. Yet, from the perspective of these subjects, some of these birds possess attributes that make them “less bird-like” than others. Abstractly speaking, such attributes that cause an entity to “drift” away from a set of central definitions are reducing that entity’s conceptual coherence relative to a category. In terms of our ontologies, adding concepts that have little relationship to the existing ones cause an ontology to become less coherent.

To measure the conceptual coherence of an ontology, we use the average distance of all geodesics between pairs of reachable nodes (where a pair of nodes is reachable if a path exists between them) [194]. A completely coherent ontology is represented as a completely connected network of concepts such that the measure is 1.0. The less coherent it is, the greater the distance between nodes. Since, under this model, average distance increases with incoherence, we use the inverse of average distance to produce a range of values that has a maximum of 1.0 for a completely coherent graph. We also multiply this value by 100 as a scaling function. Thus, our conceptual coherence metric (CCM) is the inverse of the average distance of the geodesics between all reachable pairs in a graph.

- $$CCM = \left(\frac{\sum_{i,j} dist(geodesic(i, j))}{N} \right)^{-1} * 100$$

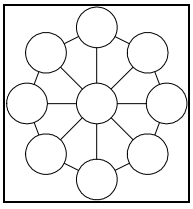


Figure 21 – Graph with average distance of 1.6, CCM = 62.5

Figure 21 shows a connected graph where the average distance is 1.6. In this graph, a central concept acts a bridge for the other nodes in the network. The nodes on the edge also have relationships with their neighbors. The combination forms a coherent structure. Figure 22 shows the same graph with the central node removed, causing the average distance to increase to 2.3.

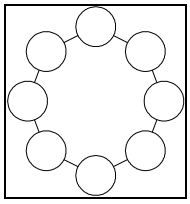


Figure 22 – Graph with average distance of 2.3, CCM = 43.5

Core concepts support other concepts by direct (aggregation and generalization) and indirect associations. Removing concepts essential to the application's domain model makes the resulting ontology less coherent, resulting in an increase in average distance and thereby decreasing its CCM.

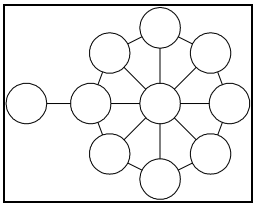


Figure 23 – Graph with average distance of 1.7, CCM = 58.8

In Figure 23, we add a node to the graph in Figure 21 to simulate a peripheral concept. In this case, the average distance increases from 1.6 to 1.7. A peripheral concept

lacking connections to central concepts reduces the overall conceptual coherence when added to an ontology.

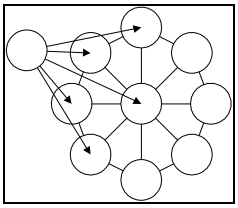


Figure 24 – Graph with average distance of 1.5, CCM = 66.7

In Figure 24, we add a node with more edges connecting it to the existing nodes, decreasing the average distance of the graph. Adding a concept that supports core concepts or is a potentially new core concept can improve the coherence of the ontology.

5.5.2 Disconnected Graphs

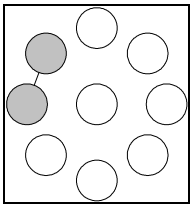


Figure 25 – Graph with average distance of 1.0, CCM = 100.0

In Figure 25, we have a mostly disconnected graph, except for one pair of nodes. Because our calculation ignores infinite distances by only including the shortest paths of reachable pairs, this graph has a CCM of 100.0. By our measure, this would be a completely coherent ontology. However, if this graph represented an actual ontology, it would be a collection of unrelated concepts, producing an incoherent ontology.

We could argue that this graph represents a degenerate case and that no developer would ever produce an application with such an ontology. However, nothing in the structure of a computing application prevents such a design. A morphology can engage operations that have no functional relationship to each other (e.g. a button that plays a DVD and another that turns on a microwave). For example, in our case studies, the Calculator / Calendar ontology consists of four isolated sub-ontologies that enable time, calendar, calculator, currency exchange calculator features. However, knowing that the Calculator / Calendar came in an "executive kit" that included a digital recorder and business card holder, we infer that these features exist in support of business activities, such as ensuring attendance at a meeting and calculating interest payments on a transaction. This understanding cannot be obtained from black-box methods alone but from studying the use context. In any case, if an ontology is found to have disconnected components, it should be flagged as a problematic ontology – lacking the necessary concepts and relationships to make it more coherent.

5.5.3 Case Studies of Conceptual Coherence Measures

We applied these metrics to the applications chosen for our case studies and found the following (Table 7):

Table 7 – Comparison of Coherence Metric

	CCM
CD Player	35.5
Calculator / Calendar	32.9
Scheduler	29.8
Notepad	21.7

The CD Player has the highest CCM value while Notepad has the lowest. This corresponds with our intuitions about these applications. From a feature perspective, CD Player just plays CDs, while Notepad, in addition to its text editing concepts, has extraneous features, such as configuration settings for printing specialized headers and footers, setting font display settings, and selecting paper sizes. Most of the Scheduler functions concern scheduling events and alerting the user. However, part of its ontology is devoted to synchronizing the data of the Palm Pilot with data stored on a personal computer, reducing its conceptual coherence. The Calculator / Calendar has high coherence because it is a collection of very simple functions. However, since it does multiple things that are not strongly related (e.g. time keeping and arithmetic calculation), it is less coherent than the CD Player.

Because the Calculator / Calendar's ontology contains several disconnected components, we next measured the individual ontologies of the Calculator / Calendar device. The results of this comparison can be found in Table 8.

Table 8 – Coherence Metrics within components of the Calculator / Calendar

	Average Distance
Calculator / Calendar (overall)	32.9
Calendar	62.5
Time	30.9
Calculator	54.9
Currency Exchange Calculator	100.0

The Currency Exchange calculator is the most coherent and the Time component is the least coherent. Again, from a feature perspective, the Currency Exchange Calculator only converts an amount using a currency exchange value while the Time component has features for telling the current time, telling that time in sixteen time zones, sounding an

alarm at a specified time, and counting down a specified number of minutes before sounding an alarm.

5.5.4 Variation Testing and Conceptual Coherence

Removing the core concepts from the ontology reduces the conceptual coherence of the application. It follows that the more important the removed concept, the more incoherent the ontology is afterwards. However, some concepts, identified as core in the ontology by their centrality values, lack the same real importance in the problem domain. Such concepts possess many subordinate attributes or sub-concepts but exist on the periphery of the ontology. For example, a concept with many attributes on the edge of the ontology appears to be highly central because of the number of shortest paths between its attribute nodes and the nodes in the ontology. A concept may also have many subtypes or aggregated concepts in its definition. From a user's perspective, that concept may be unimportant in everyday usage and should be considered a peripheral concept in the ontology.

Mutation testing uses programs that have been mutated from the original to improve test set quality [55]. Mutated programs that pass a test set highlight errors in the program or problems with the test set. Using a similar principle, we can distinguish between those core concepts that are truly important to the ontology's conceptual coherence and those that detract from it. We generate a variation of an ontology by removing a core concept, identified by its centrality measure, and any concepts or components that thereby become isolated from the main ontology. After generating an ontology variation for each core concept, we measure conceptual coherence to identify peripheral concepts – those concepts whose removal increases the conceptual coherence of the ontology.

Out of the case studies, we chose Notepad because it has the most clearly defined central purpose along with the largest set of relatively peripheral features. The overview entry in its Help Files contains the following text:

> Notepad is a basic text editor that you can use to create simple documents. The most common use for Notepad is to view or edit text (.txt) files, but many users find Notepad a simple tool for creating Web pages.

While primarily a text editor, like vi or pico on the UNIX systems, Notepad has printing and display features that do not contribute to text editing. We took the 15 core concepts identified through ontological analysis and generated 16 variations, including one ontology where we removed both the Header and Footer concepts because they both linked to the same sub-concepts. We then applied each metric to the original ontology and to the variations.

Table 9 – Notepad Variation Tests

Concept Name	# concepts	CCM
Text	79	20.1
[Configuration]	81	20.9
Footer and Header	66	21.0
Current File	81	21.5
Footer	81	21.6
Header	81	21.6
Alignment	78	21.8
[Header/Footer Code]	75	21.8
Margins	76	22.0
Font Style	77	22.1
Original	*82*	*22.3*
[Paper] Size	70	23.3
Paper	66	23.9
Script	71	24.0
Font	70	24.6
Page Setup	55	24.6
Font [Setting]	62	25.1

Table 9 shows the results of applying the average distance metric to the Notepad ontology variations. The Concept Name indicates the concept removed from the original ontology. Removing Text produced the least coherent ontology, as expected from what we know of Notepad's stated purpose. The conceptual coherence metric disambiguated those concepts that are core because of their attribute or subtype set and those that served as bridging nodes in the graph. It also did this independently of the number of nodes removed from the ontology.

Structurally speaking, concepts like Paper, Font, and Script have many subtypes and attributes, raising the centrality values of the parent concepts. However, they only have one edge connecting them to the main ontology. The concepts whose removal caused the ontology to become less coherent are central to Notepad's ontology, and all of them have direct relationships to Text. Font Style formats the text displayed in the Notepad window but otherwise has no impact on the document. As we would expect, its removal only resulted in a slight decrease in coherence from the original ontology. Conversely, the removal of Header and Footer, which possess many concepts for structuring the appearance and position of text in the header and footer of the printed document, produced a more incoherent ontology than the original.

5.5.5 Conceptual Coherence and Usefulness

Our CCM can be used to measure the conceptual coherence of an application and, when used on variations of an ontology produced by systematically removing its core concepts, can distinguish between core concepts and those peripheral concepts identified as core by their structural attributes. We claim that conceptual coherence and usefulness are related. However, CCM only provides sterile structural information about a

computing application. An application's usefulness is ultimately determined by its actual use. In order to correlate high conceptual coherence to usefulness, we developed a methodology called *use case silhouetting* described in the next chapter that enables us to measure the usefulness of an application relative to a use context.

6 USE CASE SILHOUETTES

The analysis methods presented in Chapter 5 identify core concepts and conceptual subgroups and measure the conceptual coherence of an ontology. These abstract and structural methods must be correlated to the actual or probable usefulness of the application. Without a connection between the abstract analysis and real-world behavior, we are left with many unanswered questions. For example, are core concepts in the excavated ontology the same concepts that would be central to the target domain or to the specific practices of the users? Do these subgroups have any correspondence to groups of related tasks in the use context? Does conceptual coherence have any meaningful correlation to an application's actual usefulness to a likely user.

In this chapter, we present the technique of *use case silhouetting*, a methodology for measuring the usefulness of an application to a specific use context. Use case silhouetting bridges this gap between actual usage and abstract analysis.

6.1 Usefulness and Ontological Coverage

In Chapter 1, we defined usefulness as the extent to which an application succeeds in assisting a set of users to achieve a set of goals, relative to the amount of effort required to engage those features. We also stated that usefulness is a function of the conceptual fitness of an application's ontology to the domain of the user. By measuring how well the concepts in the user's domain match those in the application's ontology, we can estimate the conceptual fitness of the application. We introduce a metric, *ontological coverage*, which measures the percentage of the ontology covered by concepts identified in a use context of interest, depending on the desired unit of analysis. For example, one could

measure the amount of ontological coverage for tasks performed by a single user, operations during a scenario, or all the actions of an organization over a period of time. A high ontological coverage indicates that the concepts invoked in the use context have a high conceptual fitness to the application ontology. Conversely, a low ontological coverage reflects a poor conceptual fitness and a probable lack of usefulness of that application to its users. An application with a high conceptual fitness is more likely to be useful to users in that particular use context than one with low conceptual fitness.

6.2 Related Work in Evaluating Usefulness

6.2.1 Software Quality

Software developers tend to focus on the engineering aspects of system development, taking the perspective that usefulness can be ensured by building what the customer requests. These requests are collected by requirements engineers and refined into requirements specifications [54, 109]. Software acceptance testing activities measures the conformity of the system design and correctness of implementation to these specifications [24]. Specifically, they measure the precision and correctness with which the system adheres to the customer's requirements using verification and validation testing activities throughout the development process. Verification activities test whether the product conforms to the specifications derived from the customer requirements and validation activities test whether developers are "building the right product" [179]. Theoretically, validation activities should ensure usefulness for the customer. However, Sommerville argues that validation activities cannot be performed on requirements specifications due to the lack of a frame of reference.

> "The main problem of requirements validation is that there is nothing against which the system can be validated. A design or a program may be validated using the specification. However, there is no way to demonstrate that a requirements specification is correct. The validation process can only increase your confidence that the specification represents a system which will meet the real needs of the system customer." [192]

Because of this lack of a reliable framework, the validation of computing applications has been limited to systems such as embedded systems that can be tested using formal methods and quantifiable testing techniques [2, 98, 214]. Formal methods can produce formal specifications that can be validated by inspection, assuming that the requirements are clear and the specifications are well organized [110]. During the requirements process, a formal method can clarify the informal statements made by customers [212] or provide reasoning techniques to identify inconsistencies or gaps in the specifications [34]. However, these formal methods assume clarity, consistency, and domain understanding from the customers and are therefore primarily verification and not validation activities. Thus, while software engineering techniques exist that can test whether the developing system conforms to the software's "blueprint", its requirements specification, no techniques exist to determine if there are fundamental gaps or errors in the specifications themselves.

6.2.2 User Interface Design and Usability Engineering

The other software development activities that attempt to ensure usefulness are user interface design and usability testing [62, 96]. Usability engineering focuses on the correspondence between the user interfaces of a computing system and the user's conceptual models of how the tasks should be performed. The techniques used by usability engineers include task analysis, user testing, iterative design, participatory

design, and prototyping [159, 160]. However, these activities simply seek to ensure that the user can access the features of the software as efficiently as possible and that the external presentation of the software matches the user's understanding of these features. Again, these are engineering concerns and assume that the functionality of the software has been precisely articulated by the end users of the system.

6.2.3 End-User Analysis

Other methods for measuring usefulness are more direct. Information Systems researchers consider usefulness in the context of *perceived usefulness* defined as "the degree to which a person believes that a particular system would enhance his or her job performance" and *perceived ease of use*, which he defines as "the degree to which a person believes that using a particular system would be free of effort" [1, 55]. Techniques for assessing the usefulness of a particular system consist of end-user interviews and surveys [1, 55, 56, 76, 120, 127].

6.3 The Use Case Silhouette

When engaging the services of a computing application through its morphology, users invoke concepts in its ontology. For example, in a word processor, selecting the "Save" menu item in a "File" menu invokes the concepts "File" and "Document". With this in mind, we can measure ontological coverage by examining the frequency by which the concepts are referenced. If we find that the set of concepts frequently referenced also correspond to the core concepts in the ontology, we could infer that the application has a high usefulness because of this relationship.

We call the overall process of recording instances of concept activation or concept frequency from a use context, *silhouetting*. The application's morphology provides

affordances that permit access to the services. Viewed another way, these morphological elements provide portals in the 'skin' of the application through which the underlying conceptual model can be seen. Activating particular elements in the morphology casts a 'silhouette' on the concepts below where only specific concepts are highlighted as we show in Figure 26.

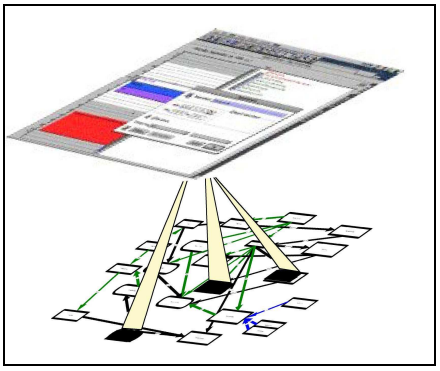


Figure 26 – The Silhouette Metaphor

We feel the appropriate unit of analysis for our work is a task description that lists the sequence of actions required to accomplish a particular goal. *Use cases* are part of the Unified Software Process (USP) [105] and describe "a sequence of actions, including variants, that a system performs to yield an observable result of value to an actor. [26]" They are related to a class of techniques in requirements engineering called scenario-based requirements engineering [110, 164, 185]. Scenarios describe a sample procedure or execution of a system by presenting specific, concrete episodes. *Use case silhouetting* is the process of developing a silhouette or an application using concepts identified in a set of use cases. By recording the number of times concepts are activated by operations on morphological elements or the number of times concepts are mentioned in a set of use cases, we can measure the ontological coverage for a proposed system. The ontological

coverage metrics obtained from a use case silhouette enable us to measure an application's conceptual fitness.

Our methodology has similarities to user interface analysis techniques such as GOMS (Goals, Operators, Methods, Selection) [42], task analysis [58], run-time behavior and system logs to track user behaviors [60, 61], sequence models [24], and cognitive walkthroughs [58, 150]. The general objective of these techniques is to identify or verify a sequence of actions that the user used to perform a task that contribute to the completion of a goal. Use case silhouetting is also related to El-Ramly and Stroulia's work on using system-level traces of user interactions to develop requirements [60, 61].

6.4 Use Case Imaging

The idea for use case silhouetting came from reading studies in cognitive neuroscience which examined the electroencephalograms [69], X-Ray computed tomography, or Magnetic Resonance Images (MRI) [168] of subjects performing certain cognitive tasks to identify those regions of the brain became active during this process. Since we were interested in identifying those concepts that became active during a task, it seemed reasonable to adopt a similar approach to identifying the active portions of an application's ontology. In cognitive neuropsychology and cognitive neuroscience, subjects are given tasks to perform designed to invoke cognitive behaviors such as visualization or memory retention and recall [64].

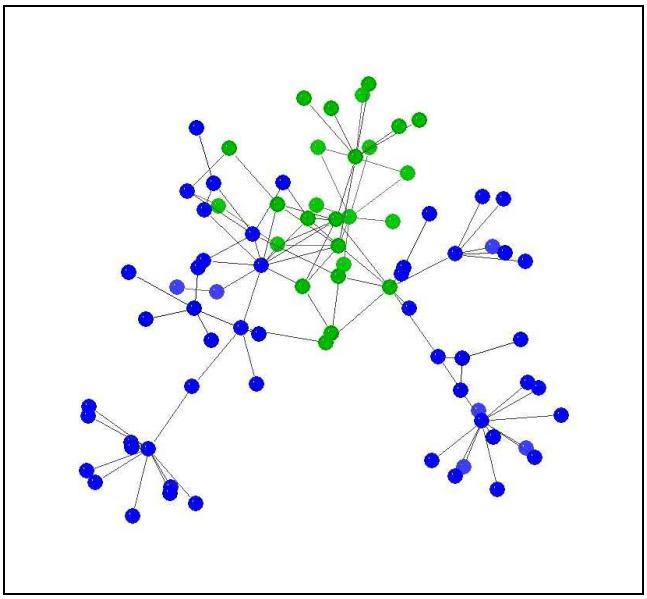


Figure 27 – Use case image for Notepad's Header and Footer use case

While we used the idea of imaging to inspire our techniques, we also borrowed the concept directly and generate images that highlight the areas in an ontology referenced by a use case or set of use cases. The diagram produced by visually representing the ontological coverage of a use case in an ontology is called a *use case image*. Figure 27 shows an example of a use case image. The green concepts are referenced in the use case describing how to edit and format Headers and Footers in Notepad. We imagine that future applications of this imaging technique would include supplementing design and analysis of computing applications and displaying dynamic animations that highlight the active portions of an ontology during the performance of specific tasks or during specific periods of time.

6.5 Overview of Use Case Silhouetting

The basic steps of use case silhouetting and analysis are:

1. Identify or develop a set of use cases – First identify or develop a set of use cases that best reflect the behaviors to be measured against the ontology.
2. Apply each use case to the ontology – For each use case and procedure within the use case, identify the relevant concepts and count the number of occurrences.
3. Measure the ontological coverage from the use case silhouette – The silhouette can be developed from the data gathered in the previous step, using basic coverage, frequency analysis, and weighted importance. From the silhouette we can calculate ontological coverage of the ontology by the use cases.

6.6 Step 1: Identify a Set of Use Cases

The first step is to identify the set of use cases that will be applied to the ontology. Use cases can be obtained from instruction manuals, requirements elicitation [165], ethnographic observation and interviewing [51, 99, 166, 178], or log files. From these sources, we can obtain scenarios [164] that describe goals and tasks [4] that a user would likely perform in a particular use context. They can even be enhanced with data describing frequency-of-use and importance of the specific task. Alternate sources, though not entirely independent, include the Help files or instruction manual of the computing applications. However, using manufacturer-supplied documentation to supply use cases might provide different results than sources obtained directly from the users. In our case studies, we found that complicated features were often given more use cases and took more steps to accomplish than simpler features likely to be used often. The completed set of use cases should consist of only those cases relevant to the use context being studied.

Here is sample text from one of the use cases of the CD Player, taken from the application's Help files (CDs: storing track titles):

> To store the track titles of your CD:
>
> Make sure your CD is in the drive.
> On the **Disc** menu, click **Edit Play List**.
> In **Artist,** type the artist's name.
> In **Title**, type the title of your CD.
> In **Available Tracks**, click the track whose name you want to store.

Not all sources describe a use case as a sequence of clearly defined procedures. In these situations, the prerequisite for inclusion in a use case set is that the source contains a goal to be achieved and includes a general description of how that goal can be reached using the computing application.

6.7 Step 2: Apply Use Case to the Ontology

Each individual use case should be analyzed for references to concepts in the ontology. How this analysis occurs depends on the form of the use case descriptions. For high-level use cases, where the interface is not mentioned, we simply examine the concepts described at each step of the use case. For example, a use case action that says "The customer requests a transaction slip from the system." tells us that the 'customer', 'transaction', and 'transaction slip' concepts have been activated in the ontology by that specific action. For low-level use cases that explicitly describe how the user interface is activated, we simply account for each morphological element mentioned in the sequence of actions and trace the concepts that they invoke. In the example above, the CD player use case references the morphological elements Disc Menu, Edit Play List Menu Item,

Disc Settings Dialog Box, Artist Text Field, Title Text Field, and Available Tracks List. From these elements we can identify the concepts Disc, Artist, Title, Track (2 times), Track Number, and Playlist (3 times). Other use cases may reference the concept names directly, making this task easier.

6.8 Step 3: Measure the Ontological Coverage from the Use Case Silhouette

6.8.1 Developing the Use Case Silhouette

A use case silhouette is developed by recording the number of times a concept is referenced in a set of use cases. This can be accomplished using two basic methods:

- Count each occurrence of a concept once within a use case. This produces both a list of concepts within a use case and the frequency of a concept across a set of use cases.
- Weigh use cases by frequency of usage or importance to user and use these weights to modify the concept counts. To study probable or actual use of an application, concepts can be weighted by the number of times users perform the use case or by the importance that users attach to achieving the goals associated with the use case. These weights can then be applied to concepts mentioned in the use case. So, a concept's occurrence can be multiplied by the weighting factor to produce a higher value.

The first method produces a basic use case silhouette. Figure 28 shows a use case image representing the ontological coverage of Notepad's help file, which is roughly 80% of the ontology. The second method produces a richer silhouette by highlighting specific areas of importance or, conversely, by highlighting areas that do not contribute to the conceptual fitness of the application to the use context. The use case image displayed in Figure 29 shows the same concepts highlighted in Figure 28 but weighted by

frequency. The red and orange nodes are the concepts Text, [Configuration], and [Current] File.

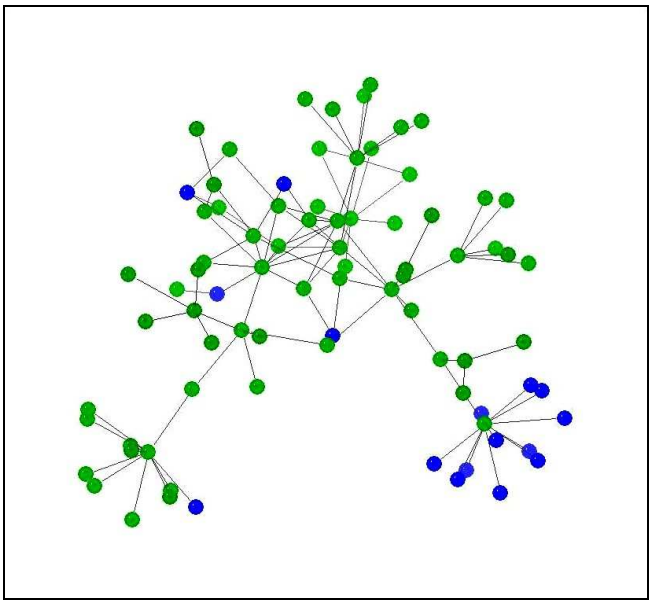


Figure 28 – Use case image showing ontological coverage of Notepad's help file

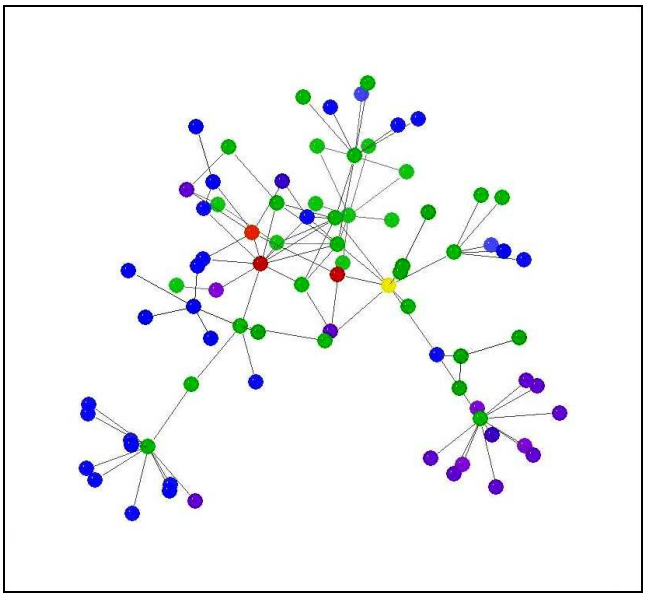


Figure 29 – Use case image showing ontological coverage of Notepad's help file weighted by frequency

6.8.2 Ontological Coverage Metrics

We can measure the following from the use case silhouette:

- *The total amount of ontological coverage provided by a set of use cases* – Assuming that the use case set provides a complete set of usages or a specific use context, we can calculate how many of the ontology's concepts are mentioned, and express it as a percentage. If ontological coverage is 100%, then the ontology has complete coverage and high conceptual fitness with respect to the domain described by the use cases. If the coverage is low, then the application does not have a high conceptual fitness for this set of use cases.
- *The amount of ontological coverage by a particular use case* – An individual use case may have low or high engagement with the application's ontology, measured by the number of concepts, especially core concepts, that it activates. A frequently used use case with high engagement must be considered carefully during design because the concepts that it uses could affect the overall ontology even if those concepts lack importance measured by centrality.
- *The importance of a particular concept relative to a set of use cases* – A concept frequently invoked by the use cases may or may not have structural importance in the ontology. If it does, the concept must be carefully designed as it has both conceptual and functional dependencies. If it does not, this means that users are referencing a peripheral concept frequency, indicating that the users are attempting to fulfill goals that the application does not properly address.

6.9 Use Case Silhouette Studies

In our case studies, we identified use cases from the help or instruction sets of each application. Because these were low-level use cases, they described each of the morphological elements that would be triggered during the use case. We used the relationships between the morphological elements and the concepts to identify the concepts silhouetted by the operations in the use case. For each application, we present

tables showing the ontological coverage of the set of use cases, some of the use cases and their metrics, and a list of the most frequently accessed concepts and their occurrence percentage relative to the total number of concepts invoked (including duplicates).

6.9.1 The CD Player

The CD Player allows the user to play CDs, to manage information about that CD, which has to be entered manually by the user, and to manage custom playlists.

Table 10 – CD Player Use Case Silhouette Statistics

Source	Help file associated with application
# of use cases:	23
# concepts invoked:	16
Total # concepts	20
Ontological coverage:	80%

Table 11 – CD Player Sample Use Cases and their Ontological Coverage

Use Case Name	**# Unique Concepts**	**% Coverage**
Adding Tracks to Play Lists	5	24 %
Deleting Tracks from Play Lists	3	14%
Moving Between Tracks	2	10 %
Options	6	29 %
Stopping a CD	1	5 %

Table 12 – CD Player Frequency of Concept appearance in use case set. Core concepts are italicized.

Name	**# Times Accessed in Use Case Set**	**% of Total # of concepts invoked**
Playlist	18	26 %
[Current Track]	10	14 %
[Current Disc]	8	11 %
Track	8	11 %
[Play Mode]	7	6 %
Artist	4	6 %
Title	3	4 %

From its measured ontological coverage, we claim that the CD Player displays relatively high conceptual fitness relative to its use cases. One possible discrepancy with regard to potential actual use of the application is the prominence of the Playlist concept relative to the concepts of Current Track and Current Disc. Because these use cases are derived from help files, more complicated features, like managing playlists, required more steps to describe, producing a larger silhouette on the ontology. The concepts not covered in the use cases concerned different play modes of the CD player, such as Continuous Order.

6.9.2 Notepad

MS Notepad is a text editor that comes with the Windows operating systems. Notepad accepts a variety of types of text files in different encodings and displays and prints a document using application settings that are applied to every text file read by Notepad. These display and print settings do not get saved with the program. It also support a Log which is a code entered on the first line of a text file so that the current day and time get printed with the document.

Table 13 – MS Notepad Use Case Silhouette Statistics

Source	Help files associated with application
# of use cases:	32
# concepts invoked:	66
Total # concepts	82
Ontological coverage:	80%

Table 14 – Notepad Sample Use Cases and their Ontological Coverage

Use Case Name	# Unique Concepts	% Coverage
Adding a Log	7	9 %
Change Page Setup	11	13 %
Changing Fonts	10	12 %
Creating Headers and Footers	25	30 %
Editing Text	2	2 %
Print Document	2	2 %

Table 15 – Notepad Frequency of Concept appearance in use case set. Core concepts are italicized.

Name	# Times Accessed in Use Case Set	% of Total # of concepts invoked
Document	16	10%
Text	15	9%
Current File	13	8%
Page Setup	12	7%
[Configuration]	4	2%
Case	4	2%
Orientation	4	2%
[Header/Footer Code]	4	2%

The use cases for Notepad also showed Notepad to have a high conceptual fitness based on its ontological coverage. The concepts not accessed by Notepad included types of paper, the Character concept, and the File concept, presumably because they were too low level to articulate in a use case. A surprising result of the Notepad Use Case silhouette is the prominence of printing features, such as setting Header and Footer parameters, and display features, such as changing the font. Both of these features are important in the silhouette but, like Playlists in the CD player, are subordinate to the main functions of Notepad which concern text editing.

6.9.3 Calculator / Calendar

The Calendar / Calculator implements an alarm clock, calendar, calculator, currency exchange calculator, and countdown timer. The clock also allows its users to view times in sixteen different time zones.

Table 16 – Calendar / Calculator Use Case Silhouette Statistics

Source	Instructions included with device
# of use cases:	11
# concepts invoked:	48
Total # concepts	48
Ontological coverage:	100%

Table 17 – Calculator / Calendar Sample Use Cases and their Ontological Coverage

Use Case Name	**# Unique Concepts**	**% Coverage**
Setting the Calendar	5	10 %
Set Count-Down Timer	4	8 %
Set Alarm	5	10 %
Calculator	11	23 %
Set Keytone On / Off	1	2 %

Table 18 – Calculator / Calendar Frequency of Concept appearance in use case set. Core concepts are italicized.

Name	**# Times Accessed**	**% of Total # of concepts invoked**
[Time Zone]	16	13%
Count Down Timer	9	8%
Hour	7	6%
Minute	7	6%
Second	7	6%
[Mathematical Operation]	6	5%

The Calendar / Calculator, according to the ontological coverage measures, shows the highest conceptual fitness to its use cases. From the list of concepts we see that the

device's primary function is timekeeping. Even so, the calculator use case has a large silhouette on the ontology, but because it encapsulates all the basic mathematical operations that one would expect to find on a basic calculator.

6.10 Discussion

Use case silhouetting provides a reasonable first approximation of an application's conceptual fitness by measuring the ontological coverage of a set of use cases. This analysis is clearly sensitive to use case selection and, to a lesser extent, the fidelity of the ontology excavated from the application. In our case studies, we obtained the use cases from the help files of the applications. The results from our studies do not likely reflect actual usage of these systems because help files are written to illustrate the application's functionality. This critique does not invalidate the potential benefits of silhouetting. Use case silhouetting can be applied to the ontology of any application from any set of use cases. Because use case silhouetting relies on tracing a sequence of activities specified by a use case to determine the concepts invoked in the ontology, it could be combined with usability testing techniques, such as those using user interface events [89].

7 CASE STUDY: POWERPOINT 2000

We have presented our methods for ontological excavation and analysis and our method of use case silhouetting for measuring the conceptual fitness of an ontology to a use context. We have illustrated how these methods work using examples from our case studies. However, we used very small applications in our case studies. Small applications, while useful for testing methodologies, tend to have small ontologies and are naturally conceptually coherent. To test our research claims and to demonstrate the scalability of our methods, we apply our methods to a large, complex, and well-known computing application: Microsoft PowerPoint.

7.1 PowerPoint 2000

Microsoft PowerPoint 2000 is primarily a tool for building and delivering presentations. According to its help file, its key features include:

- Creating presentations
- Adding drawings and graphics
- Delivering presentations
- Creating presentations for the Web
- Using the Web for working with others

While it does possess mechanisms that can be exploited to purposes other than presentation support [14], we hypothesize that PowerPoint will reveal an ontology that contains a primary cluster of core concepts related to presentations and slides and supporting concepts such as graphics and animations as smaller ontological clusters connected to it. PowerPoint should evidence a high conceptual coherence both in the structural analysis of its ontology and in its use case silhouettes.

7.2 Morphological Cartography

We first mapped the PowerPoint user interface. The morphology has sufficient complexity in navigational recursion that we chose to use the menu bar to structure our map and guide our traversal of the interface. The menus references nearly all the morphological elements, objects, and services available in PowerPoint. We included all submenus, dialog boxes, toolbars, and shortcut menus, which are menus accessed by selecting an item inserted into a presentation and pressing the right mouse button.

PowerPoint 2000 has a number of features that access external programs, such as the Visual Basic macro programming tool and Microsoft Organizational Chart. When we could clearly distinguish a window as a separate application, we modeled it with a placeholder for that application. We modeled the different views (e.g. Normal View, Slide Show Sorter, and Outline View) as modes of the Main Window of PowerPoint. We also captured all the toolbars. While PowerPoint does have keyboard shortcuts, we chose not to model these as Office 2000 provides customization functions that allows users to alter the command and menu settings.

Our survey produced a map consisting of 3267 morphological elements. When printed the map is 206 inches long and 14 inches wide. We provide a screenshot of this map at 3% magnification with a small section of it expanded to 165% magnification in Figure 30. This long and shallow morphological structure results when designers ensure usability by placing elements to be as accessible from the main window as possible.

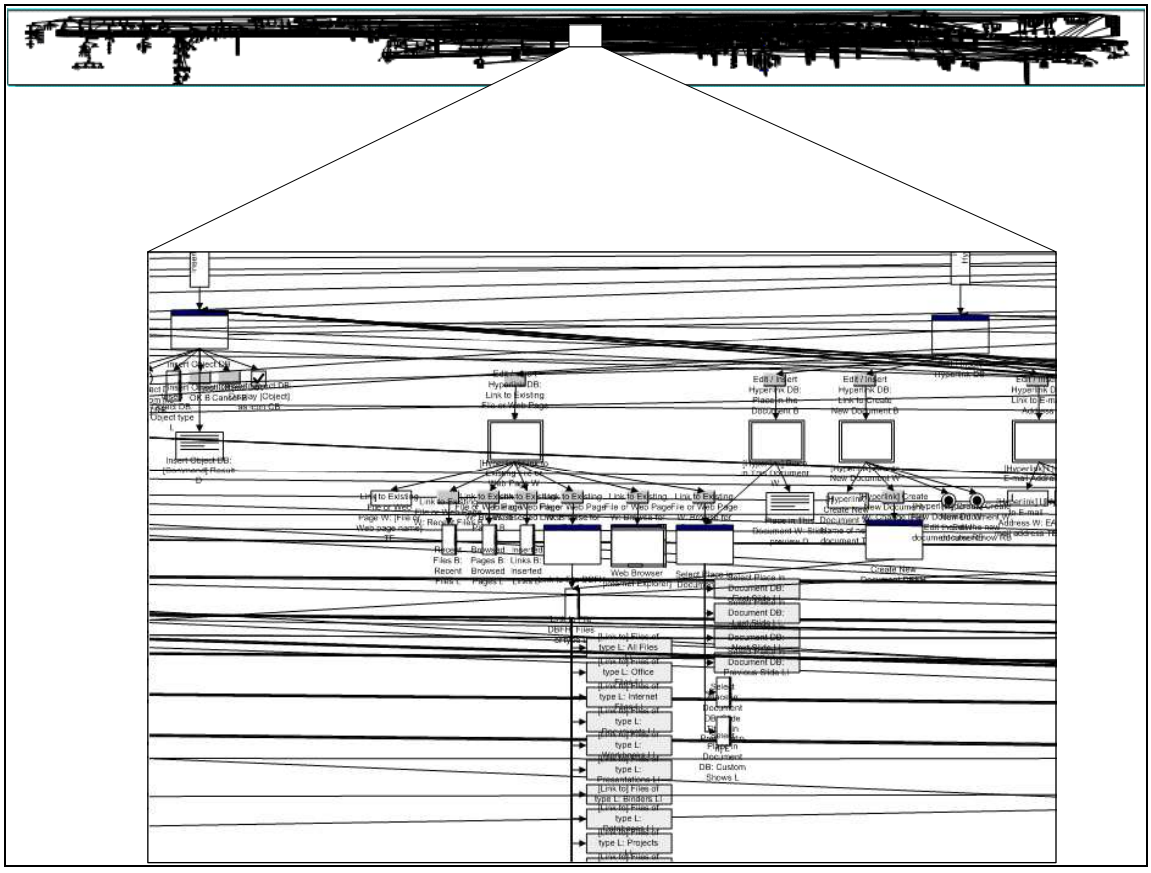


Figure 30 – PowerPoint 2000 Morphology at 3% magnification (enlarged portion at 165% magnification)

7.2.1 Distribution of Morphological Items

Not surprisingly, the majority of the morphological elements control the insertion and formatting of Slide and Notes Page Objects. PowerPoint 2000 is a workpieces style of application [102] and produces an artifact used primarily in work settings. It follows that most of the elements would be devoted to helping users insert, format, and structure items into this artifact; in this case, the presentation. The next largest group of elements manage Slide Show functions, one of the central features of PowerPoint. The rest of the elements serve supporting functions, such as editing and file management. We produced a visualization of the morphology and color coded it to highlight areas of functionality.

We show this visualization in Figure 31and show the categories of morphological elements and their color codings in Table 19. Note the entanglement between the insert and formatting functions. The long branches show deep interface traversal, such as a wizard. One of the green branches is the Genigraphics Wizard, a tool that allows a presentation to be sent to the Genigraphics company for additional formatting.

Table 19 – Legend for PowerPoint 2000 Morphology Visualization

Category	# of Elements
Insert Items	769
Format Items	692
Slide Show Items	617
File Menu Items	521
Tools Menu Items	214
Main Window	181
Edit Menu	57
View Menu	49

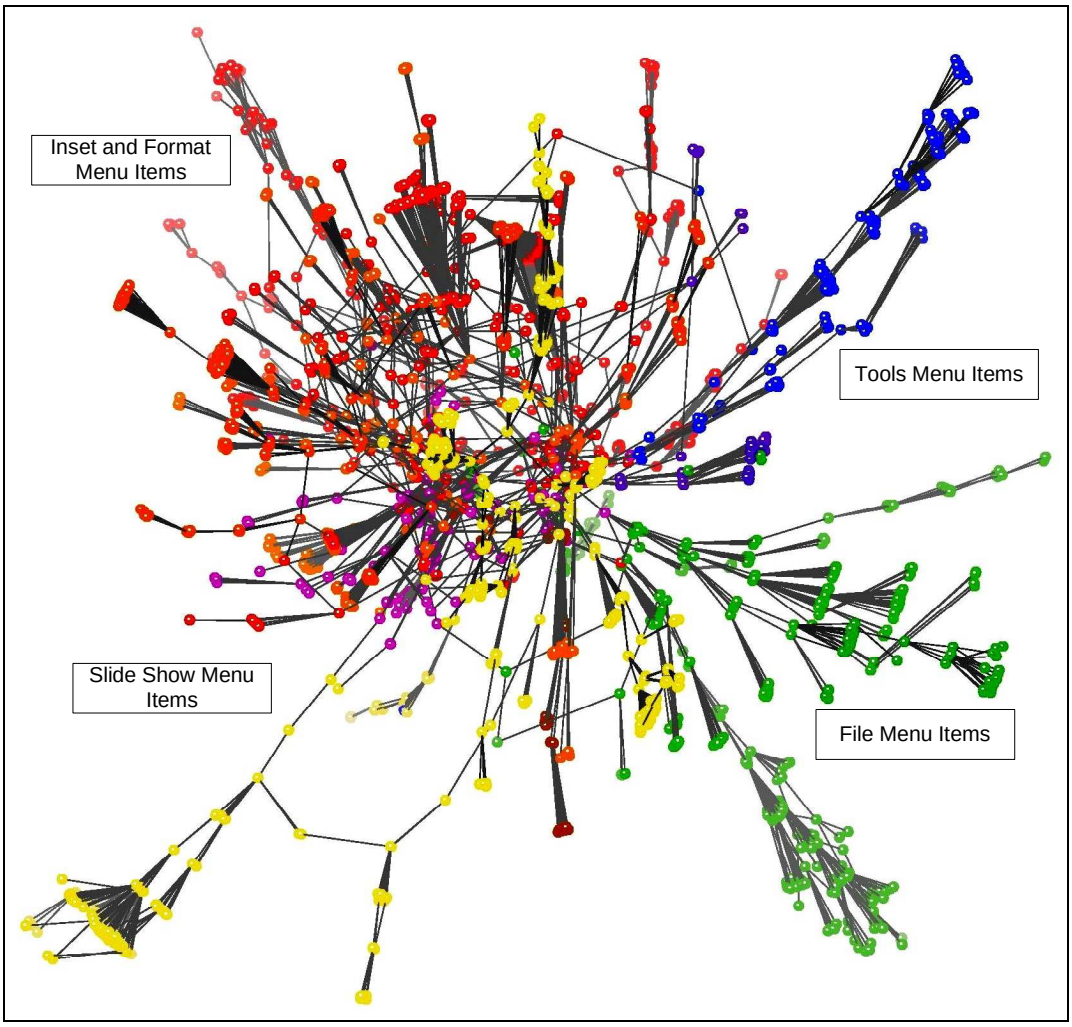


Figure 31 – PowerPoint 2000 Morphology Visualization

7.2.2 Excluded Functionality

We did not model the implementation of certain features, such as the ability to insert the Microsoft Organizational Chart or to program macros in Visual Basic. Because these applications could clearly be identified as external to PowerPoint, we did not include them in our map but left placeholders in the morphology, such as a symbol indicating an interactive object for the Microsoft Organizational Chart, to show that an object generated by this external application can be inserted into the presentation.

We did include features that we know to be shared across the Microsoft Office suite. These included the AutoShape objects and drawing operations, the ClipArt Gallery, and WordArt. Because these features have been integrated into the application's morphology, we could not distinguish them as external functionality. For this reason, we also modeled several wizards, such as the Projector Wizard, a tool for connecting the computer to a projector, and the Genigraphics Wizard, a tool designed to assist a user with ordering consulting and processing services from an external company. We also modeled one of the Internet based collaborative tools, Online Broadcast, that allows a user to broadcast a presentation over the web.

7.3 Ontology Construction

Manually assembling an ontology for an application the size of PowerPoint 2000 can be problematic due to logistical difficulties, such as planning the layout over a large diagram, and conceptual difficulties, such as ensuring that the relationships between all concepts have been accounted for. We chose to construct the ontology by building small ontological components (listed in Appendix F) using specific morphological containers to guide our organization.

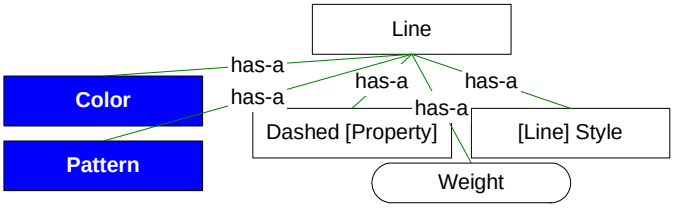


Figure 32 – The Line component

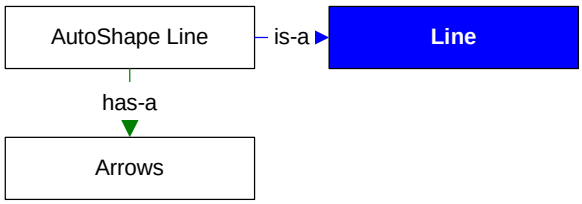


Figure 33 – The AutoShape Line component and its parent Line.

For example, Figure 32 shows the ontological component for Line and Figure 33 shows the component for the AutoShape Line. A Line is any line that can be displayed in a Presentation. An AutoShape Line is simply a line that can be drawn by the user that can also display arrowheads at either end. AutoShape Lines share all the attributes of Lines, making AutoShape Line a generalization of Line (Figure 33). We modeled this using an is-a relationship between AutoShape Line and its parent in the component. The shaded nodes serve as placeholders for edges. We assembled the ontology by inserting components and connecting all the concepts to the externally referenced components.

Figure 34 shows the PowerPoint 2000 ontology in a 21” by 30” diagram at 29% magnification. The expanded portion is at a 150% magnification. The ontology contains 1686 concepts.

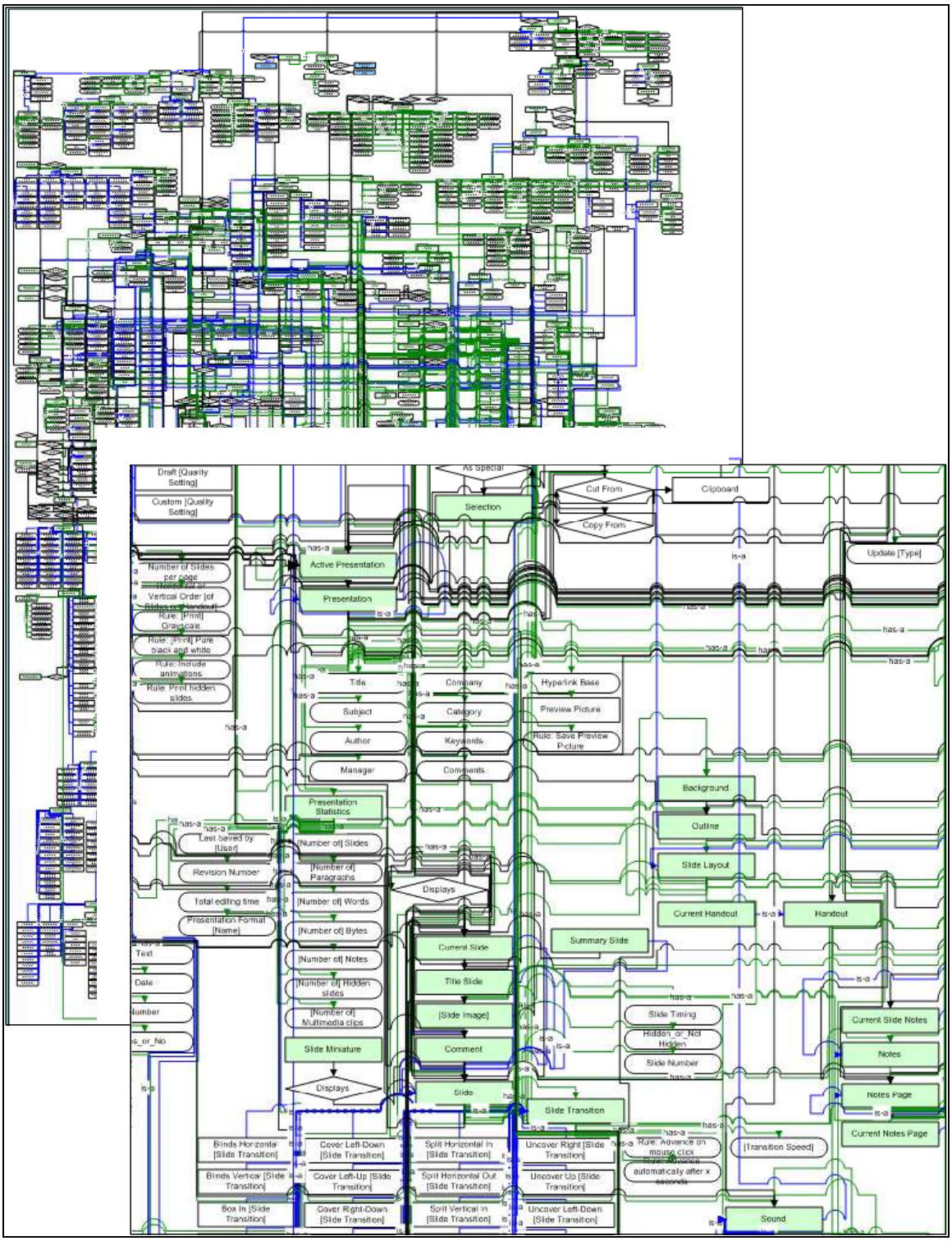


Figure 34 – The PowerPoint 2000 Ontology (29% magnification, enlarged portion 150% magnification)

7.4 Ontological Analysis

7.4.1 Core Concepts

Using our ontological analysis techniques, we identified the core concepts of PowerPoint from its ontology. The concepts reveal PowerPoint to be fundamentally a graphics application. Many of the prominent concepts are related to graphics functions, including AutoShape [Draw] Object, Line, Color, WordArt, and Animation. Other concepts are organizing ones. Presentation, Slide, Slide Show, and File are concepts that structure the objects that the users create, modify, store, and display. Slide Objects and Notes Page Objects have many of the same elements. However, because PowerPoint is a tool for generating presentations, Slide Objects have a few more properties, such as Animations, Action Settings, and Hyperlinks. [Configuration] is our constructed placeholder for organizing the option settings that modify the behavior of the application.

There are several concepts that do not belong to any presentation tool: Send To [Destination], Genigraphics Wizard, and Online Broadcast [Tool]. The Genigraphics Wizard provides many features for ordering services from a company that will refine and generate materials for a presentation. The Send To [Destination] concept supports the Genigraphics Wizard. The Online Broadcast [Tool] allows a user to display a presentation over a web page and contains numerous attributes and concepts that define how a participant can be contacted. None of these concepts directly support presentation creation but have such complexity in their subgraphs that the analysis identifies them as core.

Table 20 – Core Concepts of PowerPoint 2000

	Concept Name	Centrality Value	Description
1	AutoShape [Draw] Object	23.86	This is any AutoShape object that is not a line or a connector - this includes Text Boxes.
2	Presentation	21.40	Presentation is the work product that users create and edit in PowerPoint.
3	Slide Object	17.05	A slide object is any object that can be inserted into a slide.
4	Slide	16.38	A slide is a frame in a presentation.
5	PowerPoint File	15.44	A PowerPoint file is any file that PowerPoint recognizes.
6	Slide Show	15.33	A slide show is how a Presentation is typically viewed by an audience.
7	Color	15.02	Color - everything has color in PowerPoint.
8	Send To [Destination]	10.80	A PowerPoint file can be "sent" to an email recipient.
9	WordArt	10.00	This is a graphic that takes text and displays it with special formatting.
10	Genigraphics Wizard	9.56	This is a wizard that provides commercial consulting and packaging services for PowerPoint presentations on the user's request.
11	Text	9.51	Anything related to
12	Line	9.12	Anything that is a line in PowerPoint - usually a border for a draw object.
13	Selection	8.98	A highlighted set of text or objects or slides in PowerPoint - to be cut, copied, or pasted.
14	Online Broadcast [Tool]	8.86	Allows the user to set up a presentation through a web page for other users to log in and view during the presentation.
17	Animation	8.17	A setting that can be possessed by a slide object or text that animates it during a slide show.
18	File	8.10	A general system file.
19	Notes Page Object	7.76	Any object that can be placed on the notes page which can be printed out as a handout.
20	[Configuration]	7.48	An inferred concept encapsulating all options such as Edit and Print options.

7.4.2 Teleon Identification

We performed a *k*-core analysis on the ontology of PowerPoint. It produced a large *k*-core where *k* ranges from 2 to 7. The entire group consists of 300 nodes with no discernible relationship within the any of the *k*-cores. For example, the 7-core, consisting of nodes with 7 or more edges contained the following concepts: Appear [Preset Animation], AutoShape [Draw] Object, Camera [Preset Animation], Column Text, Dissolve [Preset Animation], Drive-In [Preset Animation], Flash Once [Preset Animation], Fly From Top [Preset Animation], Flying [Preset Animation], Media Clip, Preset Animation, Table, Text Box, Wipe Right [Preset Animation]. The 6-core had only 1 node, AutoShape Line, as did the 5-core – Split Vertical Out [Preset Animation].

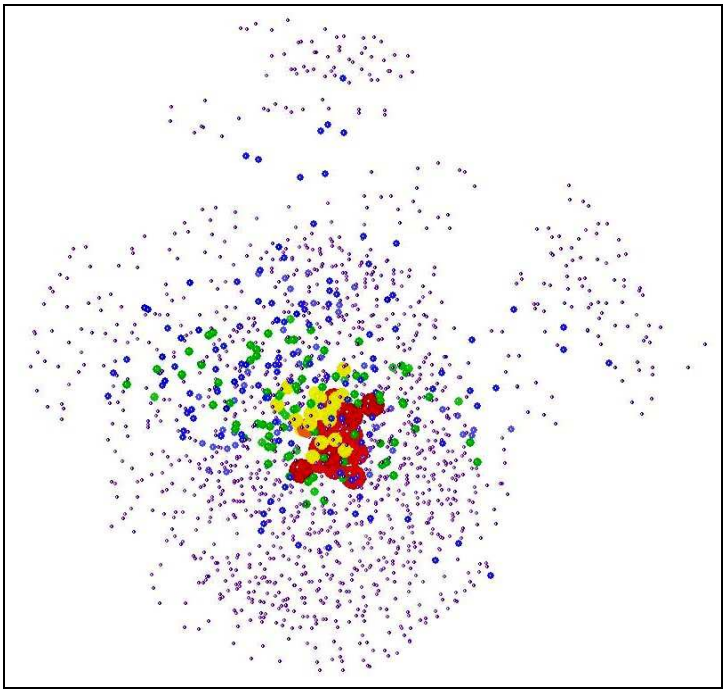


Figure 35 – *K*-Core Visualization of PowerPoint (no edges)

We provide a visualization of the *k*-core in Figure 35 where nodes were colored in intensity and size based on their *k*-value (7-core → red, 5-core→ yellow, 4-core → green, 3-core blue, 2-core and 1-core → purple).We claimed in 5.4 that a *k*-core analysis would reveal teleons that define features from the user's perspective. But this method did not work for the much larger Power Point study because it failed to identify small coherent subgroups.

In the case of PowerPoint, Preset Animation is a concept that links to several central concepts. Some of its animations only apply to specific objects, such as Charts or Text. The rest can be applied to every cell object. Because Preset Animation is a type of Animation, it ends up providing a central bridge for numerous paths, creating the 7-core shown in red in Figure 35. Interestingly, in spite of being a major part of the 7- and 6-cores, Preset Animation does not have sufficient centrality to register as a core concept. These types of constructions in the ontology seem to have no specific conceptual relationship that could be obtained from a *k*-core analysis. Thus, we could not identify any distinguishable teleons using this method.

7.4.3 Conceptual Coherence Measurements

We calculated the CCM for the ontology of PowerPoint and obtained a value of 6.9. In comparison to the other case studies, PowerPoint has a CCM that shows it to be much less coherent than the smaller applications described earlier (see Table 21). As PowerPoint implements a larger variety of features than any of the other applications, we would expect its conceptual coherence to be smaller.

Table 21 – CCM of PowerPoint and Case Study Applications

Application Name	# of Concepts	Conceptual Incoherence
PowerPoint	1686	14.56
Notepad	82	21.70
Scheduler	58	29.74
Calculator / Calendar	48	32.70
CD Player	20	35.41

This comparison shows that the CCM has some relationship to the number of concepts in the ontology but is not directly proportional to it. However, the CCM for PowerPoint is somewhat obscure in the absence of any data points from applications of similar size. We applied variation testing to the application to identify which concepts contribute to PowerPoint's coherence. We hypothesized that the concepts that we identified in 7.4.1 as not contributing to presentation generation reduce the conceptual coherence of the application.

Table 22 – CCMs for PowerPoint Variants

Concept Removed From Ontology	CCM
PowerPoint File	13.95
Slide Show	14.20
AutoShape Draw Object	14.22
Color	14.24
Slide	14.25
Text	14.30
Slide Object	14.34
File	14.35
Line	14.44
[Configuration]	14.45
Notes Page Object	14.46
Selection	14.47
Word Art	14.47
Original (All Concepts)	*14.53*
Animation	14.54
Presentation	14.56
Genigraphics Wizard	15.21
Send To [Destination]	15.25
Online Broadcast Tool	15.29

Curiously, removing Presentation improves both the coherence and complexity of the ontology. When we recalculated the core concepts using this ontology variation, we found that Slide Show replaced Presentation as the most prominent concept. One can argue that if the concept of Presentation is removed from the PowerPoint ontology, a user could still create slides, handouts, and display slide shows. Since PowerPoint has the capability of saving presentations in other formats, the "Presentation" concept serves as a general organizing concept. The variation metrics in Table 22 also show that removing the Genigraphics Wizard, Send To [Destination], and Online Broadcast Tools increases the conceptual coherence of PowerPoint. This supports our earlier contention that they are peripheral features that could be removed from PowerPoint and appear as core concepts because of the complexity of the subgraphs associated with those concepts.

7.5 Use Case Silhouetting

7.5.1 Use Case Source: *PowerPoint 2000 for Windows for Dummies*

We obtained use cases from *PowerPoint 2000 for Windows for Dummies* [129]. The *For Dummies* books are a series of popular self-help manuals written to explain things as simply as possible to the widest possible audience. In order to meet the parameters of simplicity and generality, the use cases possess less detail than a use case found in the application help file and the coverage of functionality is also less comprehensive.

In our other case studies, we used help files and instruction sheets as our source of use cases. Instructions from these sources usually describe the sequence of morphological elements needed to activate a particular feature of an application. However, we chose not to use PowerPoint 2000's help files for several reasons. First, we wanted a silhouette of the application obtained from an outsider's perspective. The writer of the *For Dummies*

books was not part of the PowerPoint development team and his main expertise is in learning and teaching the use of technology to non-technical people. Second, we found in our use cases that complex concepts in applications tend to have many use cases associated with them in an instruction manual or help file accompanying the application. Because complex concepts are often not the ones most frequently used or central to the ontology, we wanted a less technical set of use cases that would exclude every possible nuance and usage of such concepts. Third, using a source external to the application helps to add validity to our findings. For example, the set of use cases from help files will likely have 100% ontological coverage and have a concept frequency that mirrors the concepts in the ontology. However this set of use cases is unlikely to reflect any sense of actual usage of the application as they would assume a user or set of users who knew every piece of functionality in PowerPoint and used them. Another potential source is human experts familiar with these systems. However, experts have deep knowledge about an applications workings and would supply use cases that do not reflect average usage, whereas a trade paperback written for a general audience presents a set of common and important use cases that reflect the "conceptual fitness" of the application to its likely users.

7.5.2 Organization of the Source and Use Case Identification

The chapters in the *For Dummies* book each describe general areas of PowerPoint 2000 usage. For example, Chapter 2 contains the use cases and concepts related to formatting text, designing templates for presentations, and using the color and fill functions associated with drawing objects. A typical use case has a set of navigation instructions followed by some directions for performing a specific task. Use cases in the

For Dummies book often lack the traceability to a specific morphological element. However, they do have precise instructions followed by advice or suggestions for implementation. so we simply looked for key words that could be traced to concepts in the ontology. Below is an example of a use case from the For Dummies book:

> To create embossed text, follow these simple steps:
>
> 1. Highlight the text you want to emboss
> 2. Use the Format → Font command to pop up the Font dialog box. Sorry – PowerPoint has no keyboard shortcut or toolbar button for embossing. You have to do it the hard way.
> 3. Check the Emboss option.
> 4. Click the OK button.
>
> When you emboss text, PowerPoint changes the text color to the background color to enhance the embossed effect.
>
> Embossed text is hard to read in smaller font sizes. This effect is best reserved for large titles.
>
> Also, embossed text is nearly invisible with some color schemes. You may have to fiddle with the color scheme or switch templates to make the embossed text visible.

From this use case, we identified the use of the concepts “Selection”, “Font Format”, “Emboss”, “Text”, “Color”, “Font Size”, “Color Scheme”, “Title Text”, and “Template”.

We ignored sections that did not include a set of instructions for accomplishing a task. We also ignored use cases that did not have concepts that we recovered in the ontology. For example, the *For Dummies* book covers areas of file management and using the help system that may be necessary topics for someone new to Microsoft Windows functionality. It also covers the use of applications in the Microsoft Office

suite, such as the Equation Editor, which we never came across in our excavation. Other examples include Microsoft Organizational Chart, Microsoft Graph, Microsoft Chart, Microsoft Excel spreadsheets (briefly), and Tables for Microsoft Word. Obviously, the complete use case set, including the excluded items, compose an ontology much larger than that of the PowerPoint application.

7.5.3 Use Case Analysis and Conceptual Frequency

We identified 199 use cases from 299 sections in the book. The ontological coverage of these use cases is 30% (see Table 23).

Table 23 – PowerPoint 2000 Use Case Silhouette Statistics

Source	*PowerPoint 2000 for Windows for Dummies [129]*
# of use cases:	199
# concepts invoked:	499
Total # concepts	1686
Ontological coverage:	30 %

We hypothesize that a set of use cases reflecting average usage of an application invokes concepts in the ontology with a frequency that should parallel the structural importance of the concepts within the ontology. In other words, core concepts identified from the ontology are frequent concepts in a typical set of use cases. In Table 24, we list the most frequently accessed concepts in the use cases along with their centrality values and whether they have membership in the set of core concepts.

Table 24 – Partial List of Concepts Ordered by Times Referenced in Use Cases

Concept Name	Frequency	Centrality Value	Core?
Slide	48	16.38	Y
Presentation	43	21.40	Y
Text	34	9.51	Y
Selection	34	8.98	Y
Color	23	15.02	Y
AutoShape [Draw] Object	21	23.86	Y
[Current] Slide	20	2.40	
Slide View	20	1.21	
Slide Master	18	0.30	
Font [Format]	16	3.43	
File	13	8.10	Y
Position (of Slide Object)	12	0.19	
Outline View	12	0.40	
Text box	11	3.84	
Fill	11	4.43	
Active Presentation	11	5.99	
Slide Sorter View	11	0.16	
Color Scheme	10	0.93	
Slide Show	10	15.33	Y
Copy From (Clipboard)	10	0.00	
Sound	9	4.89	
Paste From (Clipboard)	9	0.00	
Outline	8	3.79	
Title Text	8	1.95	
Notes Master	8	0.27	
Paragraph	8	1.53	
Bullet	8	0.91	
Line	8	9.12	Y
Picture [Clip] (Clip Art)	8	0.09	
Normal View	8	0.11	

Many of the concepts on the list do not appear as core concepts in the ontology. Some of this can be attributed to how the use cases were written. For example, Current Slide appears frequently because a use case will start “Move to the slide” – which refers to the current slide as opposed to a more general reference to slides. In the case of the Views, many use cases have an instruction “Switch to the Slide View”. Clipboard functions – copying and pasted – were also referenced often throughout the cases as methods for placing objects.

The use case silhouette of PowerPoint does show that the Slide Master has much more importance than the structural metrics suggest. The author referenced slide masters in many ways, including creating backgrounds, preparing templates, and changing footers. We first checked to see if we had made a modeling error in our ontology. Figure 36 shows the Slide Master ontology.

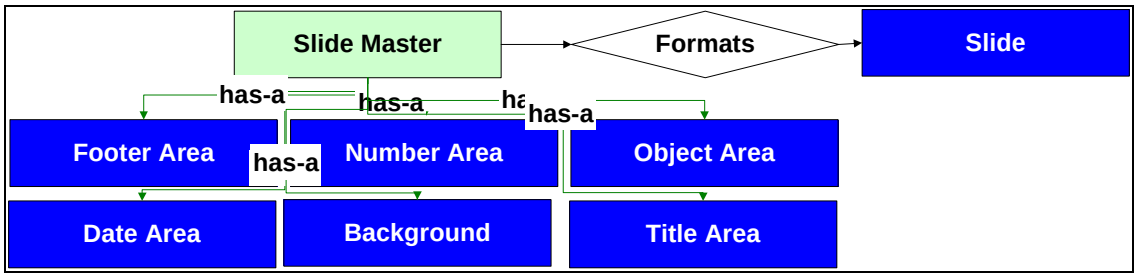


Figure 36 – The Slide Master ontology

While we support our use of an association to show the relationship of the Slide Master to Slide – that the Slide Master formats all Slides in a Presentation – this example shows a potential weakness of our representation. The Slide Master formatting affects all the slides in a presentation. Thus, a user must take extra care to ensure that the Master is designed correctly. On the other hand, this example demonstrates how use case

silhouetting can reveal potential areas of concern for developers that would not emerge in an abstract requirements analysis or testing methodology.

When we compare the top six core concepts in Table 24 to the core concepts listed in Table 20, we find the following correlations (shown in Table 25):

Table 25 – Top Six Core and Use Case Silhouette Concepts (matches italicized for emphasis)

Top Six Core Concept	**Top Six Concepts in Use Case Silhouette**
Presentation	*Slide*
AutoShape [Draw] Object	*Presentation*
Slide Object	Text
Slide	Selection
PowerPoint File	Color
Slide Show	*AutoShape [Draw] Object*

At least three of the core concepts can be found on both lists: Presentation, Slide, and AutoShape [Draw] Object. Text, Selection, and Color are also core concepts and Slide Show does appear as an important concept in the Use Cases, as do Line and File. While this supports our hypothesis, there are a number of concepts that do not appear as frequently in the use cases as their centrality values suggests. We have listed these in Table 26.

Slide Object, PowerPoint File, Notes Page Object, Animation, and [Configuration], as we discussed in Chapter 7.2.2, are partially inferred concepts that we introduced to model certain important generalizations. Because they were constructed indirectly from the PowerPoint application, most use cases did not refer to them explicitly. However, AutoShape objects, Text Boxes, and ClipArt are considered Slide and Notes Page Objects. Slide Objects have animations and action settings. Animation ties together the

concepts of Slide Transition, Entry Animation, and Preset Animation. Presentations and Slide Shows are also PowerPoint Files. [Configuration] models the possible global options that can be set in the PowerPoint Tools Menu. If we accounted for the complete generalization and aggregation in the counts of those particular concepts, they would appear with more frequency across all the use cases.

Table 26 – Core Concepts Absent from Use Case Silhouettes

Concepts Missing from Use Case Silhouette	**Centrality Value**
Slide Object	17.1
PowerPoint File	15.4
Send To [Destination]	10.9
WordArt	10.0
Genigraphics Wizard	9.6
Online Broadcast [Tool]	8.9
Broadcasts	8.6
Send To	8.3
Animation	8.2
Notes Page Object	7.8
[Configuration]	7.5

The other missing concepts confirm what we had proposed earlier and confirmed with variation testing – that the features whose removal from PowerPoint 2000 improved its conceptual coherence are peripheral to the application, despite their centrality values. Online Broadcast, the Genigraphics Wizard, and WordArt all have sufficiently large ontologies that they appear as core concepts in the ontology. Now, a use case silhouette consisting of a set of ‘likely’ uses of the application show that these are not very important to users. One use case is devoted to each topic, on average. If the same analysis were performed using actual user data, these concepts may disappear entirely from the analysis.

7.5.4 Use Case Silhouette Visualization.

To emphasize our finding that the concept frequency in a use case silhouette mirrors the centrality of the same concepts in ontological analysis, we show visualizations of the core concepts and the use case silhouette in Figure 37 and Figure 38, respectively. We used the same visualization on both diagrams, coloring and sizing nodes by relative importance to their respective data sets. We also kept the same orientation for both diagrams for easier comparison. The centrality graph shows more “average value” nodes (colored green and yellow), as one would expect from a set of concepts weighted with equal importance with respect to their position in a graph. The use case silhouette mirrors the centrality visualization in the center but also shows that the peripheral branches (Online Broadcast and Genigraphics Wizard in particular) have no value in the silhouette.

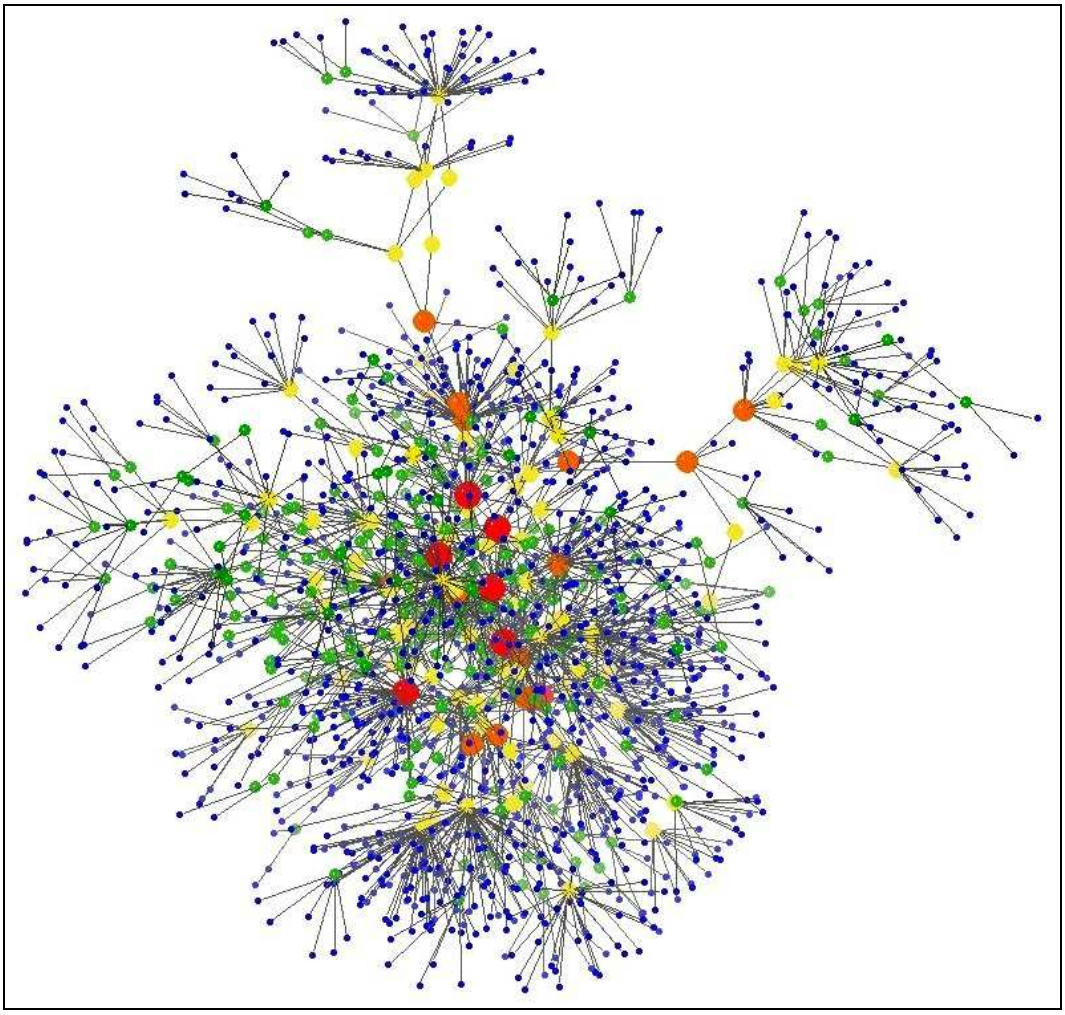


Figure 37 – Centrality Visualization of PowerPoint 2000

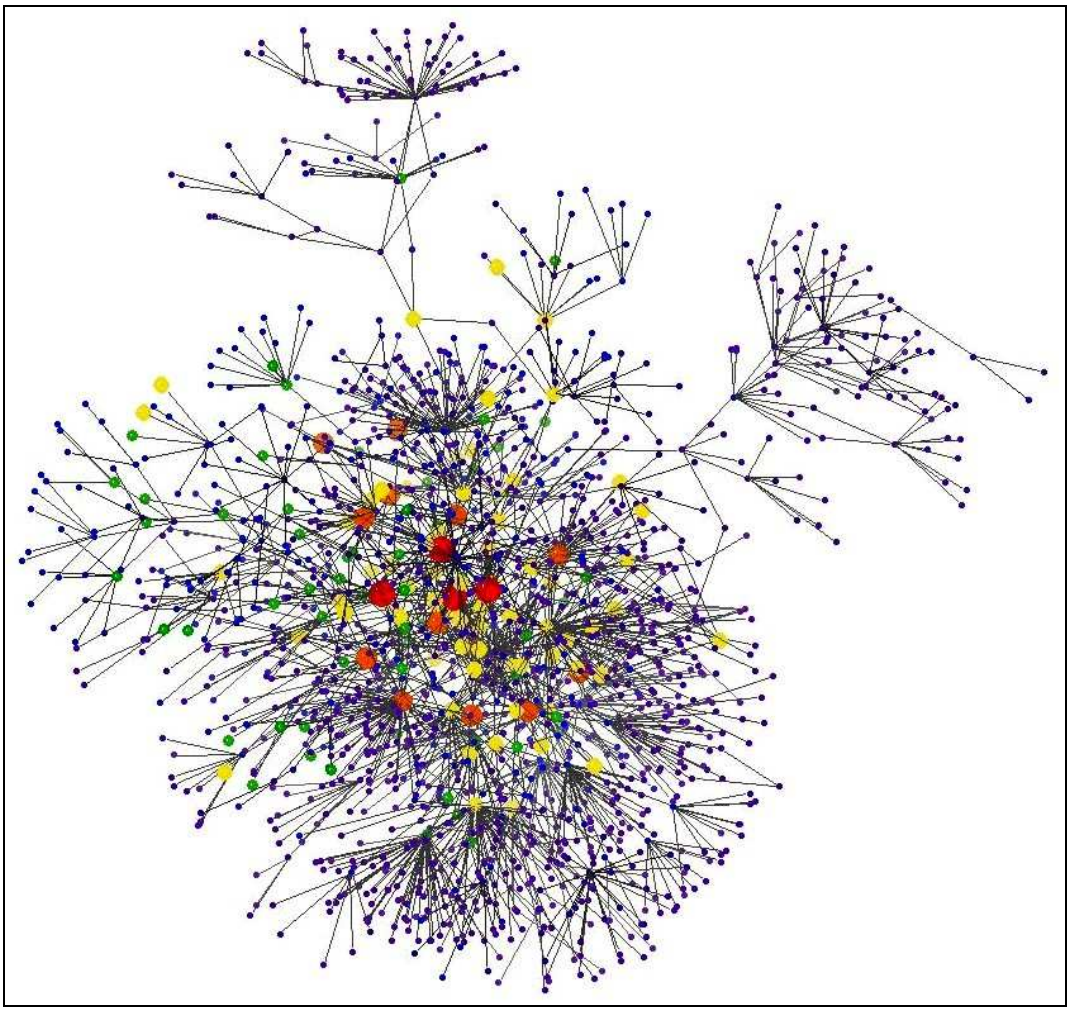


Figure 38 – Use Case visualization of PowerPoint 2000

7.6 Discussion

We have shown that ontological excavation and analysis can be scaled to large applications. We identified core concepts that belong to an application designed to support presentation creation and management. We identified peripheral features, the Genigraphics Wizard and Online Broadcast, using variation testing. Our use case silhouette, using a self-help manual, confirms that many of the core concepts identified by our structural analysis have high correspondence to those concepts referenced most frequently in the use cases. PowerPoint 2000 with the exception of certain features appears to be conceptually coherent as suggested by the visualization but not necessarily by the CCM. We believe that more studies on applications of similar size and complexity will be necessary to determine what values signify high conceptual coherence. Visually, PowerPoint's ontology, using a spring-embedded algorithm, appears as a large central ball of related concepts with some branching outliers, suggesting the intuition that the majority of PowerPoint's concepts are organized around the core presentation concepts.

If it can be shown that removing peripheral features not only improves conceptual coherence but improves usefulness without compromising marketability, then an obvious alteration to PowerPoint 2000's ontology is to remove highly peripheral features and treat them as external applications. Alternatively, if the designers feel that these features are necessary and essential given the requirements of their customers, the concepts defining those peripheral features need to be tied much closer to the application. We will discuss these heuristics further in Chapter 1.

8 CASE STUDY: MICROSOFT WORD 97

8.1 Introduction

We have shown how use case silhouetting can analyze an application's conceptual integrity and its relationship to usefulness. We have argued that those core concepts identified using our structural measures are those concepts most frequently referenced by a set of use cases. However, we have only provided examples using use cases from arguably artificial sources. Help files and self-help guides are written by people with technical expertise and expert-level knowledge about the applications. Thus, one could argue that we have simply demonstrated that our use case sources were well-written and complete in their ontological coverage. Would use cases obtained from real users reveal the same correspondence between core concepts in the ontology and frequent concepts in the use context? How can these techniques be applied to real user data?

In the case study presented in this chapter, we first performed a partial excavation on Microsoft Word 97, using the data obtained from an external researcher. That study was conducted to understand user perceptions, familiarity, and satisfaction with Word's first-level features. Using this data, we produce a weighted use case silhouette and demonstrate that a correspondence exists between core concepts and user preferences.

8.2 Capturing Users' Experience of Complex Software

Dr. Joanna McGrenere conducted a study to examine exactly how users perceived and experienced Word 97 as an example of complex software [137]. She was investigating whether users actually perceive complex software as having too many unnecessary features and how their experiences affect these perceptions. To constrain her

analysis, she based her study on *functions*, which she defines as "action possibilities (affordances) that are specified visually to the user [137]." She presented 53 participants with screenshots of Word consisting of only those first-level functions. This study used the following heuristics to define a first-level function:

- Each final menu item in the menus from the menu toolbar counted as one function. A menu item was not considered final if it produced a cascading menu (as shown in Figure 39).
- Each item on a toolbar counted as one function. Drop-down menus on those toolbars were not counted as additional functions.
- Selectable items on the status bar were counted as one function.

Using these heuristics, she identified 265 first-level functions on the default interface. She then showed her subjects with screen captures of these functions and asked them what the function did and whether they used it (see Figure 39). These responses were scored on a 2-point scale – familiar and unfamiliar – and a 3-point scale – used regularly, used irregularly, and not used. We used her screen captures to perform an ontological excavation and analysis of Microsoft Word, limiting this work to her first-level functions. We then used the data from her questionnaires to perform a weighted use case silhouette of the ontology.

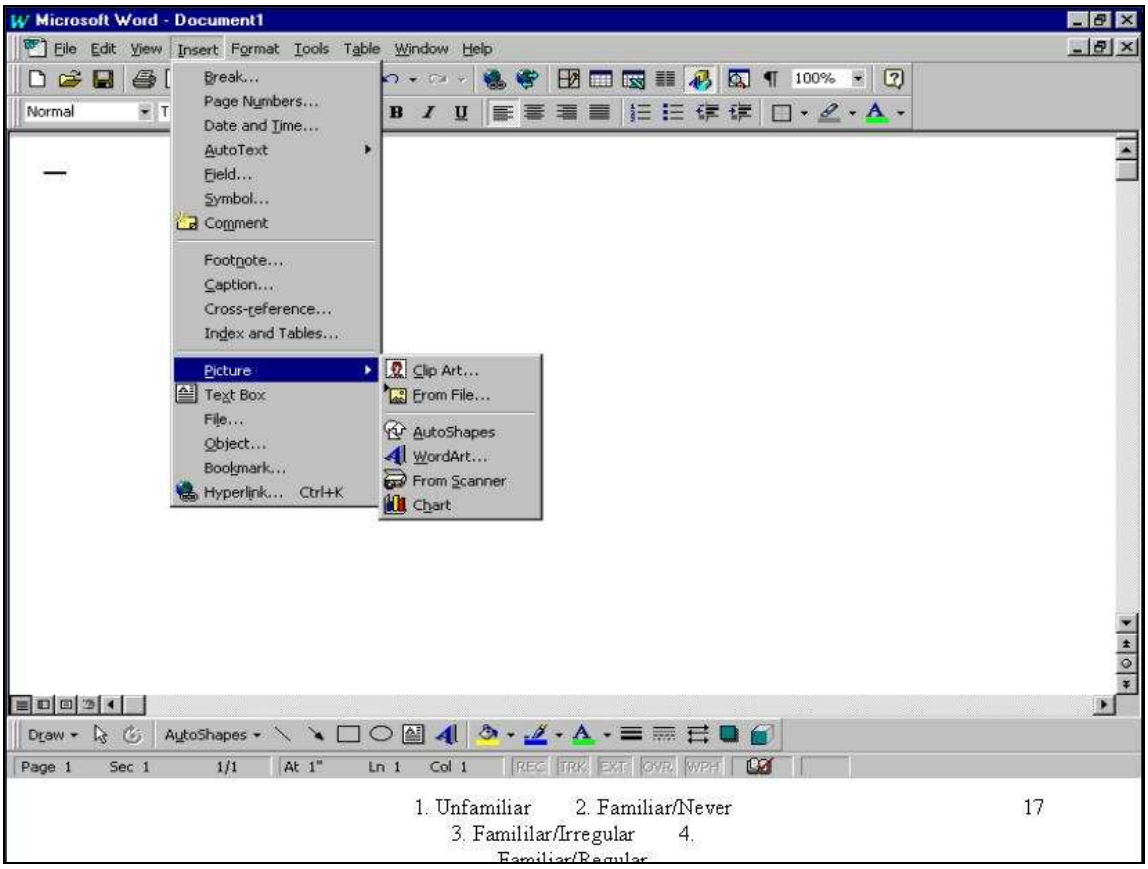


Figure 39 – A sample screen from McGrenere's Microsoft Word Study

8.3 Morphological Cartography

Since we chose to study only those snapshots of the morphology provided to users in the study, we decided not to build a complete map of Word. Instead we use the information in the screenshots to identify Word's concepts.

8.4 Ontological Excavation

Using the observable labels of the morphological elements in the screenshots, we excavated concepts. We ignored any system-level functions, such as those related to be file handling. Using just the screenshots, we identified 246 concepts (entity types and attributes). We then incorporated those concepts inferred by the screenshots. The study

focused primarily on functions provided by Word to support document creation. The screenshots did not focus on the concepts that define what a document is or what can go into one. For example, the Insert Menu does not have a menu item for Paragraph or Page, although it does have an item for inserting an explicit Page Break. A user simply types in text and the application automatically creates paragraphs and pages. We excavated another 101 concepts required to define the basics of word processing and text.

We then identified relationships between the concepts by using the morphological containers. When those structures were not sufficient, we identified relationships by dynamically interacting with those morphological elements in the application. For example, documents in Word are structured around paragraphs. Most objects that can be inserted into a document are placed into a paragraph. This cannot be determined simply by looking at static labels in the screenshots of morphological containers. We did ignore relationships that could not be inferred from the screenshots. For example, a Picture can be contained by a Page, as a floating object, or by a paragraph, using the Picture Layout type, “In Line With Text”. These concepts and relationships can not be determined from the menu items. Without any customizations, formatting a picture’s layout can only be accessed by clicking on a picture with the right mouse button to access the “Floating Picture” or “Inline Picture” Shortcut menus.

The excavated ontology for Word is small, lacking the same detail as the one excavated in our PowerPoint study, despite the fact that both applications have large feature sets and complex objects. Many of the attributes and concepts that would have been identified from looking at second-level (or third-level) functions were not included.

Nevertheless, the first level functions still provide a basic ontology consisting of concepts that we would expect to find in word processing.

8.5 Ontological Analysis

We first identified the core concepts from our excavated ontology. Table 27 lists the core concepts of the Word ontology. The concepts Document, Paragraph, and Text are the most central in the ontology; a result that we would expect to find in a word processor. Likewise, Page, Font [Format], Character, Color, and Word also make intuitive sense in the context of an application designed to produce textual documents. Tables have their own menu and have many operations and properties that define them. Fields are automatically updating pieces of text that can be inserted into a document and have relationships to other concepts in the document, such as Text.

Table 27 – Core Concepts for Microsoft Word

	Concept Name	Centrality Value	Description
1	Document	42.65	The name of the work product produced by Word.
2	Paragraph	37.08	A block of text ending in a carriage return or break.
3	AutoText	27.00	A piece of text that can be automatically inserted into a document.
4	Text	25.47	A sequence of character and special characters.
5	[Current] Document	24.46	The document currently being edited by a user.
6	Table	12.33	A structure that displays text and objects in a grid of rows and columns.
7	Page	12.27	A section of document that can be printed to one sheet of paper.
8	Field	10.62	An object that displays and automatically updates some type of text.
9	Font [Format]	9.60	A setting for displaying characters on a printed page.
10	Character	9.56	The basic unit of text.
11	Closing [AutoText]	7.38	The category of AutoText that contains closings.
12	Color	7.19	The color setting for a document object.
13	Word	7.07	A block of text preceded by and ending in a special character or break.

AutoText is found in the Insert Menu and allows the user to insert words into a document, such as those found in a Salutation (“dear mom and dad”) or a Reference Line (“in reply to”). AutoText shows up as a core concept in the word processor because the entire subgraph connected to AutoText contains 45 nodes, about 13% of the ontology (shown in Figure 40). Closing [AutoText] contains 13 nodes. We are certain that if all the functions beyond the first level and the attributes of the objects were exhaustively captured then AutoText would be less prominent in the ontology. AutoText, while related to the concept of Text, is a peripheral concept to word processing, something that was verified by our variation testing.

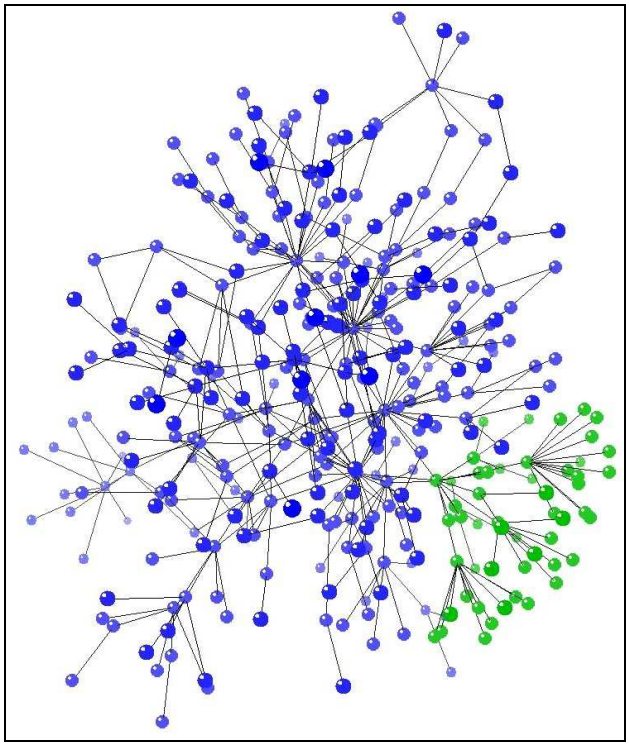


Figure 40 – Visualization of AutoText Subgraph in Microsoft Word 2000 Ontology

When we performed our variation tests, we found that removing the concepts that belong to word processing produced ontologies with a lower conceptual coherence than

the original (Table 28). As we expected, removing AutoText improved the conceptual coherence of the ontology. However, we were surprised by the CCM that showed removing Font [Format] improves the conceptual coherence of Word. It seemed odd that Font [Format] appeared to be a more peripheral concept than AutoText considering that formatting the font settings of characters is ubiquitous to most text editors and word processors today.

Table 28 – CCMs for Word 97 Variants

Concept Removed From Ontology	**CCM**
Document	17.16
Paragraph	18.32
Text	18.63
Page	19.29
[Current] Document	19.32
Character	19.33
Table	19.33
Color	19.39
Word	19.39
Field	19.63
Original	*19.64*
AutoText_Closing	19.74
AutoText	20.02
Font_[Format]	20.12

From the perspective of a word processor, the appearance of a character on a screen or on a printed page has no effect on most of the other features. Spell checkers and grammar checkers look at text but ignore all the font settings associated with those individual characters. Features, such as, paragraph alignment, pagination, column formatting, and table cell orientations are affected by the character size and typeface, which can change paragraph and page layout, but these effects happen at the system level and are not modeled part of the ontology. None of those features are affected by the font

color or style. The only other major function that uses font formatting is the Style Organizer. Thus, Font [Format] can be treated as a peripheral feature. However, users may have an entirely different view of peripheral and core functions.

8.6 Use Case Silhouetting

McGrenere collected both quantitative and qualitative data from her subjects on their familiarity with Word. We used the quantitative data to produce a use case silhouette of the Word's functions, weighting them by combining the familiarity and frequency-of-use scales. We hypothesized that the most familiar and frequently used concepts will correspond to the core concepts identified by our methods.

We first identified the associated concepts, treating each screenshot shown to subjects as a separate use case. We created the silhouette by assigning the total of the combined ratings from all subjects to the relevant concepts in the screenshot. The total possible rating a concept can receive is 212 (a maximum rating of four from each of the 53 participants). We present a partial silhouette in Table 29, including all the concepts with a rating over 90 and the ratings of all the core concepts.

Table 29 – Partial Use Case Silhouette for Word 97. Core Concepts highlighted

Concept	Combined Rating	Centrality Value
Center [Alignment]	148	0.00
Size (Pts)	148	0.00
Bold [Font Style]	146	0.00
Font [Format]	*145*	*9.60*
Font [Typeface]	145	0.86
Left [Alignment]	143	0.00
Italic [Font Style]	143	0.00
[Current] Document	*140*	*24.46*
Underline	140	0.00
Justify [Alignment]	134	0.00
Right [Alignment]	134	0.00
Printer	129	0.00
Document	*125*	*42.65*
Text	*124*	*25.47*
Word	*124*	*7.07*
Spelling [Tool]	124	0.44
Header	120	0.00
Footer	120	0.06
Page Setup	118	0.00
Zoom Setting	117	0.00
Bullet	114	0.00
Page Number	114	0.00
Print Preview	114	0.00
Numbering	110	0.00
Paragraph	*98*	*37.08*
Thesaurus [Tool]	98	0.00
Selection	97	0.60
Find / Replace Tool	94	1.16
Table	*92*	*12.33*
Page	*74*	*12.27*
Character	*70*	*9.56*
Color	*39*	*7.19*
Field	*30*	*10.62*
AutoText	*23*	*27.00*
Closing [AutoText]	*11*	*7.39*

Our silhouette shows some of the core concepts receiving high ratings amongst the users. It also shows that AutoText and Closing [AutoText], identified as a peripheral concepts in our variation tests, did not receive high ratings by users. However, other core concepts – Page, Character, Color, and Field – also received low ratings. In addition, many of the highly rated concepts are related to formatting fonts or setting paragraph alignment, both peripheral concepts. We believe that this may be explained, in part, by the methodology used for data gathering. In the study users were asked whether they were familiar with a function and how often they used it. Concepts that define Word 97, such as Paragraph, Page, Character, Color, and Field are implicit to the behavior of the application. Thus, users are less likely to perceive them as functions or features of an application. For example, page numbers that automatically update as an application is edited, are fields. Users gave high weights to the concept of Page Number but not to Field. If we were to extend our interpretation of the ratings a step further, concepts such as Paragraph or Character would be weighted higher. After all, the concepts of a Center Alignment or Bold [Font Style] would be meaningless without the concepts of Paragraph or Character.

There are two important observations that emerge from this use case silhouette. First, use case silhouetting, when combined with user data, can highlight areas of concern for a user population. These concerns may be independent of the centrality values expressed by the underlying ontology. Designers could use ontological coverage metrics to make parts of the morphology more accessible to the users.

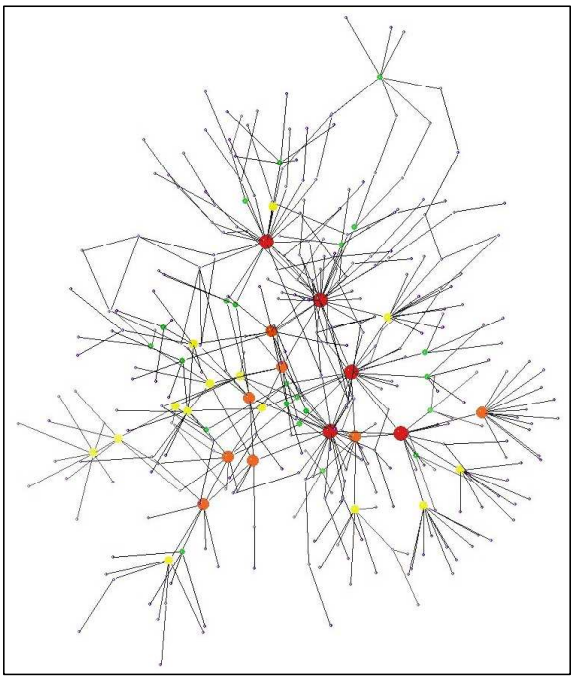


Figure 41 – Use Case Image of Word 97's Core Concepts

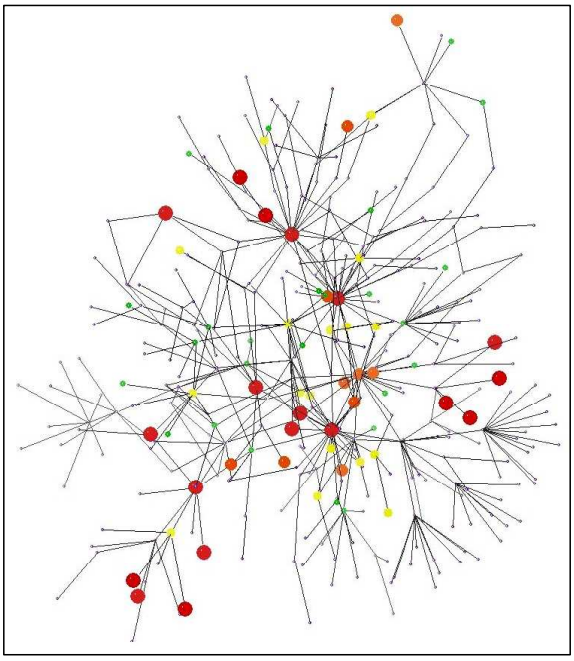


Figure 42 – Use Case Image of Weighted Use Case Silhouette

Second, users may not express or understand the fundamental concepts required to define the functions or features that they value. Designers must be careful to identify these core concepts in designing an application, regardless of whether or not the users expressed them during requirements elicitation. To highlight these differences, in Figure 41, we present a use case image of Word 97's ontology colored by intensity (red nodes have high centrality) and sized based on each node's centrality values. In Figure 42, using the same coding scheme to color and size the nodes, we present the combined ratings presented by the users in the Word 97 study. The nodes in the use case silhouette highlight frequently used and known concepts, which are much more numerous than the core concepts in the ontology.

Analyzing both centrality values and ontological coverage metrics for an ontology highlights concepts that designers should attend to carefully. When combined with variation testing, we can identify concepts and components within the ontology that are detracting from the ontology's conceptual coherence. In this case, if we remove the concepts with the lowest ratings, including AutoText, Word 97's CCM increases from 19.64 to 21.52. This is not a large increase but, nevertheless, we see that removing peripheral concepts does increase the conceptual coherence of an application.

8.7 Discussion

We have shown how weighted data from users can be used to produce a use case silhouette of our excavated ontology. We found that a direct correspondence existed between our core concepts and those concepts identified as highly rated from the use cases in the Word 97 study. However, the ontological coverage of the frequently used

concepts in the study's use cases reflected a greater emphasis on unit functions than on the core concepts.

The core concepts would have shown more prominence if we had allowed for the second-order associations in our analysis – Font Settings informing the concept of Character, for example. A tool that supports such analysis would include an application *superpositioned graph* or *supergraph*. This is a graph that consists of both the morphological map and the ontology and included the direct and indirect connections between morphological elements and concepts. Thus, a traversal through a series of morphological elements would directly highlight the relevant concepts. Using this supergraph, a use case silhouette representing the actual use of the application could be obtained using log files of user interactions with the application. The frequency of concept references could be surveyed over any period of time and number of users to measure the ontological coverage of the application. This analysis could be enhanced by a dynamic use case image of the ontology, displaying the areas of intensity and interest as they were activated.

9 CONCEPTUAL INTEGRITY AND SOFTWARE EVOLUTION

In our introduction, we suggested that applications, as they acquire peripheral features, lose conceptual integrity. In this chapter, we discuss this process of software evolution, review previous work in this area, and present the results of a case study of Microsoft Word's evolution that provides evidence of decreasing conceptual coherence in its feature set. We conclude this chapter with some lessons applicable to both designing applications and evolving existing versions of applications.

9.1 Software Evolution and Feature Aggregation

Software evolution refers to the process of growth and change over the lifetime of the software during its maintenance phase. Perry characterizes these changes into three categories [157]:

- corrections – repairs to errors in the code
- improvements – optimizations to performance, usability, maintainability, and so on.
- enhancements – additions of new features, generally visible to the users of the system.

Software tends to go through many iterations of development and enhancement, evolving over time as dictated by the competitive demands of the marketplace and in accordance with Lehman's Law of Software Evolution that states that a computing application (specifically what he calls an E-type program) "must be continually adapted else it becomes progressively less satisfactory" [119]. Ultimately, software must satisfy its users whether its role is to entertain, to facilitate intellectual activities, or to produce

work products. Because these goals are embedded in real world contexts, software engineers must contend with two issues in the development and evolution of these systems.

First, specifying and designing such systems so that the embedded domain model has sufficient fidelity to operationalize the services desired by the customers and users of the system is an immensely complicated process for requirements analyses [18, 51, 97, 102, 125, 163, 166]. Second, the real world also changes over time. Organizational goals change as do procedures and processes. Introducing new technologies also perturb the original domain as users learn and adapt their own behaviors to these new tools [101, 171]. These changes cause the software's model of the world to fall out of step with the actual world [120]. The first two categories of evolutionary changes, corrections and enhancements, simply improve an application's ability to implement its current domain model. To change that model requires that it be enhanced. Usually, this enhancement is accomplished by adding features to the application [35, 121, 135, 197]. We call this process *feature aggregation* although it is also called, more critically, *feature creep* and *creeping featurism* [154].

9.2 What is a Feature?

"Feature" has different meanings depending on the perspective or stage of software development that its used. At the requirements stage, features are clustering of individual requirements that describe a "cohesive, identifiable unit of functionality." [187, 188] or part of a specification that "a user perceives as having a self-contained functional role." [82]. Developers view features as simpler units of functionality [134]. For example, Cusumano and Selby – in describing Microsoft's culture – say the following:

> The features in Microsoft products are relatively independent units of functionality visible to end users. They are like building blocks, especially for applications products. Examples are printing, automatically selecting a column of numbers and adding them, or providing an interface to a particular vendor's hardware device. Features in systems products, such as Windows NT or Windows 95, are often less visible to the end user; Microsoft and other companies sometimes simply call these 'functions'. [49]

In system development, features are sometimes perceived as "packages of incrementally added functionality", describing how feature enhancements are added in stages to a system.[31, 39, 40].

Several software development techniques use features as their unit of development. These include feature engineering [187, 188], Feature-Oriented Domain Analysis (FODA) [76, 111], Feature-Oriented Reuse Method (FORM) [112], Feature Oriented Programming (FOP) [21, 22], and generative programming [50]. The general structure of these methods is to identify features in the problem domain, refine the concepts expressed by these features, and develop the supporting design and architectures around these features.

Product lines and product families use features as an organizing principle for development. One paper defines a product line as having a reusable infrastructure of shared behaviors and services and allows the construction of many family members [57]. Another paper defines product families as "sets of products that share architectural properties, features, code, components, middleware, or requirements. [117]" While these terms seem to be interchangeable or depend on granularity, features are used in both cases to develop a shared infrastructure for reuse [117, 131, 186].

9.3 Features and Usefulness

Does software evolve to become more useful over time? Do these enhancements improve its usefulness? An engineering expectation might be that they do. Artifacts such as forks, pencils, paper clips, and bookcases are adapted over time until they have a stable set of optimized features that allow them to perform their function well [158, 160-162]. Forks develop extra tines, pencils acquired a wood casing around their lead interiors, paper clips changed in shape and length, and bookcases develop movable shelves. Large and seemingly immutable structures such as buildings are adapted and improved over time to meet the needs of their inhabitants [33]. Even structures that are not inherently adaptable, like bridges, will see new design improvements with each new construction as technology improves and engineers learn from the failures of past efforts [159]. Thus, in engineering disciplines, development techniques improve over time, and later generations have better designs and more functional stability than earlier ones. Software is adapted to optimize its functions, but it also adds more features as it evolves – something that is difficult to do to a physical construct. Adding features allows each successive version to perform more functions and gives its users more services. From a consumer's point of view, one could argue that given the choice between two equivalently priced versions of software, the one with more features will be more attractive because of its potential usefulness. What is unclear is whether this type of evolution, driven by a combination of industrial, marketing, and consumer pressures, has truly made computing applications more useful to their users.

There have been studies to suggest that this form of software evolution does not necessarily produce a new version with increased fitness. A seminal study conducted by

Lehman and Belady on the IBM OS360 showed that as that the system aged, it becomes less stable [120]. This decrease in stability made the software more difficult to maintain as its code became more complex. Lehman's Laws of Software Evolution argue that programs must continue to grow in functionality to maintain user satisfaction. At later stages in their evolution, they become more difficult to enhance because of their growing complexity [119, 120]. Thus, from a software engineering perspective, adding features becomes a two-edged sword in that features have to be added but they add to the complexity of the system making it increasingly more difficult to adapt and improve.

From the user's perspective, at a certain point in an application's evolution, as has been noted in both the academic and popular literature, its user population begins to complain about the difficulty they have with the latest version [13, 71, 136, 139]. These difficulties include applications having too many features, automated features that are not desired, and problems with navigating the user interfaces to find the desired features [34, 118, 150, 153, 154]. Users describe such systems as *bloated*. We formally define *bloat* as *the description applied to applications when it possesses a disproportionate number of unnecessary features that interfere with normal or desired interactions with the application*. The application has lost conceptual fitness by embodying more concepts than are desired by its users.

Evolving computing applications by adding features can also result in difficulties for developers. Researchers in telephony have identified what they call the *feature interaction problem* where proposed features contradict or interfere with existing features [31, 40, 198]. This reflects a tension between the changing services desired by the customer and the established ontology of the software as encoded by developers.

Techniques are being devised to accommodate or reduce the introduction of features which conflict with existing functionality [39, 116]. Nevertheless, this problem seems to hint that application ontologies have limits to their growth at least as far as usefulness is concerned.

These computing applications do not become more useful over time, and improvements to their usefulness have different costs and implications than one might expect from then engineering or architecting of a physical construct. Evolving programs in such a manner implies an entropic process as the system decreases in stability over its lifetime [174]. Nevertheless, by studying software ontologies, we may learn what makes them prone to entropy and decreased stability. Specifically, if software is becoming less stable as features are added to them, we may be able to detect this process by studying the *diachronic variation* (variation over time) of its concepts and to examine applications that have been through several generations to see whether they exhibit increased complexity and decreased coherence in their ontologies.

9.4 Previous Work in Software Evolution

In addition to the seminal work of Lehman and Belady, there have been many similar studies of software evolution, tracking changes to system elements such as source code, number of modules, and overall stability [7, 17, 38, 72, 75, 113]. There have also been many studies showing how software should or could be evolved to achieve goals such as greater stability, fewer errors, and improved maintainability [10, 11, 18, 44, 50, 107, 108, 130, 144, 155, 156]. While this research has contributed to our understanding of software's internal composition as it is adapted over time, we still know very little about how enhancements to software affect the users of the system.

A study by Godfrey on the evolution of the Linux system did track growth by the major subsystems but studied this through lines of code rather than changes to functionality or ontology [72]. Lehman is currently developing a theory of software evolution that accounts for feature evolution through feedback loops in the global software process [122-124]. The closest work to our area of interest is a study of telephony conducted by Antón and Potts [5, 6]. They reported that the evolution of telephony features could be characterized by discrete bursts of service aggregation. They found that while normal growth emphasized the core aspects of telephony, the basic communication services enabled by a telephone, later additions included services that attempt to address "the inadequacy of, interactions among, or inventive abuse of earlier services." The expansions they detail suggest that the telephony ontology was decreasing in conceptual coherence over time. However, this was not a focus of their work.

9.5 The Feature Evolution of Microsoft Word

In studying the ideas of ontology and the evolution of an application's conceptual coherence, we wish to answer the following questions:

- How do computing applications evolve their features over time?
- How does the evolution of a computing application affect its perceived usefulness?

Work in design evolution by Henri Petroski show that tools evolve and improve over many iterations through combinations of design failure, optimization, and cultural co-evolution [159, 160]. We could make the general claim that "all tools improve with each successful version." However, software lacks the physical constraints and single-minded

design of the artifacts studied by Petroski. Thus, we need to study the service evolution of a computing application to learn what happens.

In our introduction, we described a common problem in computing application development – that of balancing market demand for additional features in successive versions of an application and the conceptual integrity of the application. Microsoft Word is the word processor bundled with Microsoft Office's productivity suite. It is a well-known and contentious examples of bloated software by both academic and commercial communities [71, 136, 138, 139]. Since beginning life as a relatively novel application that implemented word processing technologies in a graphics environment, Word has acquired features for desktop publishing, web page generation, and online collaboration, along with numerous supporting features for text editing and document creation, such as graphics and grammar checking. Despite its strong commercial success and wide familiarity amongst users of computing technology, we have found that merely mentioning Microsoft Word will elicit equal parts of compliments about Word's overall usefulness and complaints about its complexity and usability

We analyzed three versions of Microsoft Word (MS Word 2.0, MS Word 95, MS Word 97) using the kinds of objects that could be edited using the application or inserted into a document as our unit of analysis [95]. We identified them from the Insert menus of each application. In our analysis, we discovered the following:

- Word's morphology increased in depth and complexity over time. However, these changes were driven primarily by changes and enhancements to the objects. The number of operations also increased over time. While this also correlated to the number of objects added to each version, there were no discernable patterns to how this occurred.

- Objects from previous versions remained the same despite changes to the overall object view. Objects did not disappear from a new version with one or two exceptions (in which they migrated outside the application). Thus, older objects become more entrenched over time and develop more operations and morphological elements that activate them.
- New objects and corresponding services were added in "clumps" to the periphery of the object view. Rather than an even pattern of growth where existing objects acquired attributes and types and grew contiguously, like the annular rings of a tree, new objects with new concepts and operations were added to the previous set. The list of objects identified in Word 2.0, Word 95, and Word 97 can be found in Table 30.

From these findings, we arrived at the following conclusions:

- MS Word evolved most noticeably by adding new features to the previous version's set.
- The user interface or morphology is an inadequate point of analysis for understanding an application's complexity or usefulness. Morphologies are driven by the application's underlying theory. Morphological complexity can affect ease of access or activation of certain operations, but the application's overall usefulness is determined by the concepts it contains and implements.
- Older features may remain because they define the application or in order to preserve compatibility with other applications. In MS Word, if a fundamental word processing concept such as "word" or "paragraph" were to disappear in the next version, then it would no longer be a word processor.
- 'Bloat' results when adding newer features interferes with access to older features.

Table 30 – Conceptual Evolution of the Document Objects in MS Word

Word 2.0	**New Objects in Word 95**	**New Objects in Word 97**
Annotation	Caption	3D Direction
Border	Cross-Reference	3D Lighting
Character	Database	3D Object
Column	Drawing	3D Surface
Document	Drawing Object	Comment
Envelope	Font	Font Animation
Field	Font Effects	HTML Document
Font Style	Font Underline	OCX Object
Footer	Form Field	
Footnote	Heading	
Frame	List	
Header	Note	
Index	Numbering	
Line Numbering	Revisions	
Object	Table of Authorities	
Page	Table of Figures	
Paragraph		
Picture		
Section		
Shading		
Style		
Summary Info		
Tab Alignment		
Table		
Table Cell		
Table of Contents		
Tabs		
Word		

Finding that older features persisted in Word confirms that all applications, new or evolved, have an ontological foundation composed of concepts necessary for the definition of those applications. Our findings also suggest that Word's conceptual coherence is decreasing with each successive version. By Word 97, none of the objects seem integral to the activity of word processing. In fact, by the last version, Word has acquired features for web page editing. Intuitively, we know that the concepts associated with this activity are numerous and complex enough that they significantly detract from the conceptual integrity of Word. Users in 1997 may have requested these functions in large numbers, and the developers may have simply been responding to customer demand. However, we also found an increase in morphological complexity that has reduced the overall usability of the application.

9.6 Feature Aggregation and Morphological Complexity

Changes to an application's morphology need not alter the ontology of a product. Conversely, changes to the ontology can require changes to the morphology. These changes do not have a 1:1 correspondence as we found in the MS Word evolutionary record. Our analysis of MS Word shows a large growth in morphology over successive versions and only a small growth in the number of new objects. Some of these changes result from attempts to improve accessibility to existing and frequently used objects. However, most of this growth can be attributed to a basic relationship between an object (or underlying concept) and a morphological element.

Consider the simple example of a single object, with one user-level operation, accessed by a single morphological element shown in Figure 43.

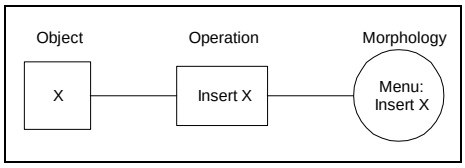


Figure 43. 1:1:1 correspondence

Many objects in an application tend to have attributes, options, and capabilities, each of which requires a function to use it properly. If the user wants to change a Font Style from Normal to Bold, an extra function is needed. This situation is better portrayed by Figure 44 than the simple correspondences of Figure 43.

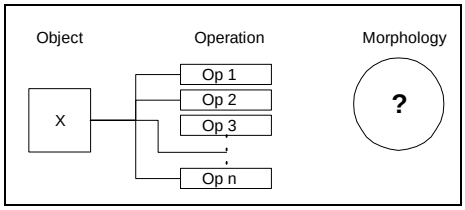


Figure 44. 1:n correspondence between object and operations.

But in order for these operations to be useful, they require some form of access from the system morphology. Important or frequently used operations may also require multiple elements to increase accessibility. Figure 45 shows how the final morphology grows from adding a new object along with multiple morphological elements to support the new operations.

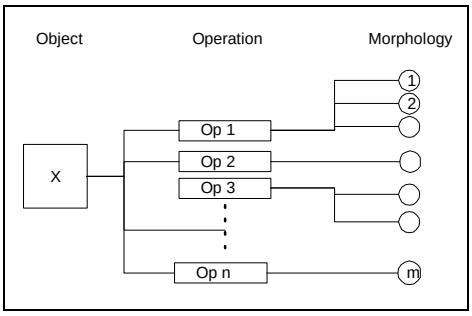


Figure 45. 1:n:m correspondences with object in system

Features often contain not just one object but several objects and tools. This illustration shows how introducing or extending a features can have tremendous impacts on the overall morphological complexity of the system. The rapid structural changes in the morphology of Word compared with the relative stability of its core features reinforces the standard architecture guideline to decouple user interface code from application features.

9.7 Evolving the Features of Computing Applications

A major practical consideration for developers is how to manage the design and architecture of a version to allow for the coherent evolution of its features. Developers planning to evolve systems need to design and structure architectures to support such coherent growth. If the objective is to preserve the conceptual integrity of an application while maintaining its competitiveness in the marketplace, developers must consider how these changes will impact the existing ontology.

Older features tend to be more entangled with associations and therefore require more effort to modify in later releases. New features with conceptual relationships to existing features also require careful design to reduce the possibility of unintended feature interactions. Because of the inherent difficulty with making changes to entrenched

features, developers may be tempted to supply new features that only loosely associate with old features and are thus peripheral to the core ontology of the application. In the initial versions of the application, these peripheral features may serve to support specific core concepts and, over time, may become integrated into the application during periods of retrenchment [5]. However, peripheral features added to support other peripheral features or that provide only tangential services to the core features of the application detract from the application's conceptual integrity while increasing morphological complexity. Such arbitrary expansions results in an application that is less useful, more difficult to learn, and harder to use.

10 DISCUSSION OF FINDINGS

We have presented our methods for the ontological excavation and analysis of computing applications and shown how they can be used to measure an application's conceptual integrity and probable usefulness. While promising, there are a number of issues regarding the accuracy and thoroughness of our methods that we have not yet addressed. In this section, we will address potential concerns with the work, discuss the significance of our findings, and summarize our contributions.

10.1 Validity and Repeatability

Campbell and Stanley described several threats to validity in qualitative research methods [41]. If ignored, these threats can influence the data gathering process such that the conclusions are suspect. A primary theme amongst these threats is the introduction of human error and bias, whether in the selection of subjects or the implementation of the methods used to study them. In our work, we have identified two central threats to the validity of our results: human error and systematic bias in model construction.

In Figure 46, we show a flow chart of our methodologies and the artifacts produced by them, representing methodologies in boxes and artifacts in ovals. The most critical artifact produced by our methods is our model of the application ontology. We have highlighted the methodologies most prone to human error and bias. Ontological analysis and use case silhouetting use analytical and mathematical methods and are not sources of threats to validity. Because the bolded methodologies depend highly on the ontology, one could reasonably ask whether we unconsciously or consciously manipulated the data modeling in such a way that we engineered results that met our conceptions of these

applications. For example, do Presentation and Slide appear as core concepts due to systematic modeling bias or are they truly core concepts of PowerPoint?

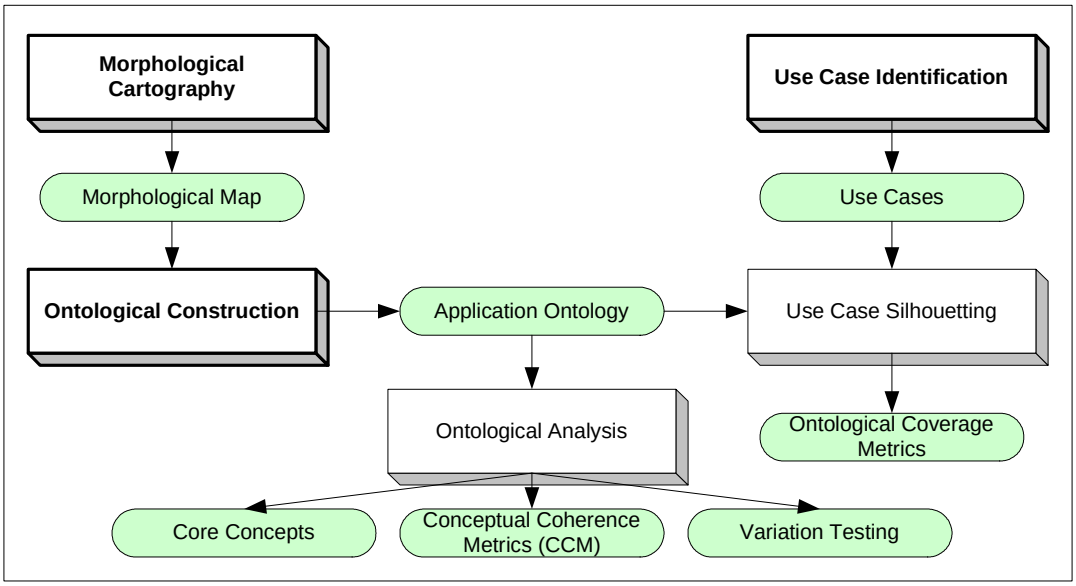


Figure 46 – Error-Prone Processes (bolded) in Ontological Excavation and Analysis

These questions about validity also feed into questions of repeatability. That is, could another person, using the methods that we have described in this paper, produce the same models and conclusions?

10.1.1 Error and Systematic Bias in Black-Box Reverse Engineering

Because we use black box reverse engineering techniques and because we have not automated them, we have few guarantees against errors in our analysis, such as overlooking morphological elements or relationships between concepts. To address the problem of model validity, we developed our methodologies to generate relatively simple representations and methods that favor abstraction over fidelity. Where possible, we ensured that the heuristics used to derive these models reduced the number of decisions

that needed to be made about how something should be modeled. Nevertheless, by adopting a black-box perspective to limit the recovered ontology to those user-visible concepts and by not seeking automated solutions, we have introduced threats to validity. We now describe how these threats are manifested and argue how we addressed them for the each stage of the process.

10.1.2 Threats to Validity in Morphological Cartography

In the case of the morphological map, we use a simple framework that allows us to capture and model the user-accessible elements in a systematic manner. Because we primarily use the map to generate the list of morphological element labels for the next stage, ontological construction, we are primarily concerned with completeness of coverage. Thus, the greatest source of error at this stage is the manual traversal of the morphology and ensuring that all the elements have been identified and mapped.

For example, we discovered a menu item in PowerPoint 2000's user interface that we overlooked – Notes Layout. It allows the user to change the layout settings of the Notes Page and appears when the user is editing a Notes Page and goes to the Format Menu. In all other modes of operation, the menu item in the same location says Slide Layout. These kinds of errors are very difficult to avoid, especially in a complex application that has many modes of operation. However, in such large applications, the internal redundancy of the morphological elements, designed to improve usability by making certain objects and services accessible from many different parts of the morphology, ensures that a concept will likely be identified somewhere in the morphology. In the case of the above error, the Notes Layout dialog box contained checkboxes for displaying the Slide Image, Body, and a way to reapply the Notes Master

settings to that page. We had already identified these concepts elsewhere in the morphology.

We argue that as long as we have captured most of the morphology, missing only a few items, that the resulting list will still contain the essential concepts of interest to our analysis. So for reverse-engineering purposes, it is reasonable to expect that someone following our methods diligently will produce similar maps for the applications we studied.

10.1.3 Threats to Validity in Ontological Construction

In the case of ontological construction, we use established data modeling methodologies to identify our concepts and the relationships between them [15, 25, 26]. For example, we identify concepts by looking for nouns in the morphological elements, ignoring those nouns that reference system-level or morphological concepts. By using this simple heuristic, we reduce concept identification to a mechanical activity. When we found it necessary to substitute our own interpretations for the sake of specificity or clarity, we made certain that our notation showed these explicit alterations to the language of the morphology. Given the same list of morphological elements, we expect that another person could derive this basic list of concepts, with some variations on attributes, inferred concepts, and naming. There may also be slight variations about what features should be represented as tools or parts of an application configuration.

More questionable are the methods by which we derived the relationships in the ontology. Many of the static relationships, such as generalizations and aggregations, could often be derived directly from the dialog boxes and containers of the application. For example, a list of items could be modeled as a parent with a list of subtypes; a series

of checkboxes or radio button in a dialog box suggest attributes belonging to the subject of the dialog box.

We derived some static relationships by interacting with the application. For example, in the Word 97 ontology, the Document contains Pages and Paragraphs. We initially asked ourselves if Pages contain Paragraphs? This relationship cannot be identified by looking at the static morphological elements. The only way to determine this, using black-box methods, is to type in some paragraphs and see what happens. Because we found that paragraphs can span pages, in the absence of any other visible organizing concept, we had to assume that Paragraphs were structured by the Document.

In the case of containment relationships, database modeling practice and other areas of software design have developed well-accepted methods for disambiguating potential structures and recognizing the existence of structure clashes (e.g. documents consisting of a stream of paragraphs and a stream of pages, but with the page structure independent of the paragraph structure) [103]. Thus, different analyses of an application by different researchers or practitioners are likely to be the result, at least in part, of the incorrect or incomplete application of modeling heuristics by some but not by others.

Association relationships present a different threat to validity. The inferential process we used to derive these relationships is the one most likely to produce deviations in independently derived models. Unlike the case of containment relations, a residue of subjectivity seems to be unavoidable. in fact it has been recognized in the data modeling community for decades [114] that there are alternate ways of modeling the same ontology fragment in terms of its relations, and no absolute criterion for choosing among the alternatives. In the absence of exhaustive and analytical heuristics for characterizing all

possible associations, we have no way of removing this particular threat to validity. Examples from future empirical studies coupled with source code analysis may reduce this variability.

10.1.4 Threats to Validity in Ontological Analysis

Ontological analysis is entirely algorithmic and mathematical. While there may be errors in how we interpret the results of the analyses, the numerical values generated for centrality, conceptual coherence, and variation testing are not questionable. We have shown in previous work that core concept identification is resistant to small errors in a graph [96]. By definition, core concepts have many dependencies on them from other concepts in the ontology. If only one or two of these dependencies were missed during ontological construction, then it is still likely that a core concept would be identified by its centrality values during analysis. However, systematic bias during the modeling process can still produce an ontology with radically different results and an argument could be made that we biased our models towards certain concepts that we believed to be central to the system.

For the largest ontology we have excavated, Microsoft PowerPoint 2000, we first composed the 1686 concepts identified during morphological cartography into 198 ontological units before assembling them into a large connected semantic network. These units ranged from one to two concepts to fifty or more, depending on their complexity, number of attributes, and number of subtypes. They were each built using specific morphological containers to structure them. A few were constructed using our interpretations. Because we built the resulting ontology atomically, the number and complexity of the concepts and units, make it difficult to introduce bias into the model.

10.1.5 Threats to Validity in Use Case Silhouetting

Use case silhouetting measures ontological coverage of a set of use cases to the ontology of an application. The threats to validity at this stage can be found in the selection of and composition of the use cases.

With regards to selection, a set of use cases can be filtered to produce an inaccurately high ontological coverage of an application to a use context. In most of our case studies, we relied primarily on sources that were written with expert knowledge of the system. Not surprisingly, we produced use case silhouettes with high ontological coverage measurements. However, we acknowledged the bias in the sources we chose and presented a subsequent case study using independently-gathered use cases derived from ordinary users. We have also argued that our methods can be validated with different sets of use cases or log data from actual user behaviors. In order to avoid selection bias in use case silhouetting, a complete set of use cases, covering all application features, should be used.

With regards to composition, a set of use cases can be composed at very different levels of detail, leaving room for interpretation. In the least arguable case, a set of use cases that describe explicit paths through the morphology to achieve a goal has the most traceability because each element invokes one or more specific concepts. In another case, a use case may be written to describe vague activities that must be performed to achieve a particular result. In the worst case, the use case may also use different language than that of the application, requiring additional interpretation to tie use case operations to the relevant concepts. In these cases, we believe that a large number of use cases will eventually support our central claim that the frequency of references to concepts should

parallel their centrality values in the ontology. Nevertheless, care should be taken to ensure that the use case terminology is applied consistently to the matching concepts in the ontology.

10.2 Evaluating Our Dissertation Claims

We now discuss the results of our studies in the context of the claims made in our introduction:

- Any computing application has a central or core set of concepts that are essential to that application's ontology and can be identified through analytical means.
- Concepts that are not essential to an application's ontology either exist to support core concepts or are peripheral to the ontology. Peripheral concepts reduce an application's conceptual coherence.
- Usage of the application will invoke core concepts more frequently than peripheral concepts in the ontology.

10.2.1 Core Concept Identification

In our case studies, we have shown that ontological excavation and analysis can identify core concepts in an application's ontology. However, we have not demonstrated that our techniques have identified all of the essential concepts. Part of the uncertainty lies in our arbitrary choice of cutoff points (a centrality value of 7.0) when determining whether a concept is core or not. The real problem is the lack of objective criteria with which to compare our findings. It is conceivable that, due to slight modeling errors or an erroneous cutoff value, we have ignored several core concepts in our case studies. While the ontological coverage metrics obtained from the use case silhouettes offered some verification for our core concepts, we also found several cases where the metrics

identified other concepts as important to the use context. Nevertheless, we acknowledge that this objective criteria may not exist and that even experts do not necessarily agree on what is essential and what is extraneous. We are confident that our methods offer an analytical means of identifying the specific concepts that define an ontology.

10.2.2 Teleon Analysis

We coined the term *teleon* to refer to small but coherent ontological units and proposed that they could be detected using a k-core analysis. We reasoned that certain concepts have natural affinities and relationships that could be detected in the structure in the form of mathematically derived components, such as cliques. In our Notepad case study, we were able to use a *k*-core analysis to identify related clusters. However, in the Palm Pilot Scheduler and PowerPoint 2000, the *k*-core analysis only revealed a large cluster of concepts tied together structurally but with no discernible relationships between them. We now believe that the *k*-core analysis is the wrong method for identifying such subgroups and an alternative method must be found.

10.2.3 Measuring Conceptual Coherence

We have developed a measurement for the conceptual coherence of an application as a first approximation of conceptual integrity. Within sets of comparably-sized and related ontologies, such as those generated for variation testing, the CCMs can distinguish between ontologies with peripheral concepts and those without peripheral concepts. In fact, using a combination of use case silhouetting with ontological analysis, we have shown that peripheral concepts, in both usage and analysis, do detract from the conceptual coherence of an application's ontology.

While we believe that the CCM does provide an adequate measure for distinguishing peripheral concepts from core concepts, a reasonable question to ask is, "What do these numbers mean?" For example, should a designer tweak an ontology to bring its CCM to a value of 30.00 ± .1? Are these values related to the number of concepts needed to express the problem domain embodied by the application? Are certain domains inherently incoherent, such that any application built to address that space will naturally possess a low CCM?

Table 31 – CCM of PowerPoint 2000 and Case Study Applications

Application Name	**# of Concepts**	**CCM**
PowerPoint 2000	1686	14.56
Word 97	347	19.67
Notepad	82	21.70
Scheduler	58	29.74
Calculator / Calendar	48	32.70
CD Player	20	35.41

Table 31 shows that applications with smaller numbers of features have higher CCM values. However, we believe PowerPoint to be a conceptually coherent application. With a few exceptions, all of PowerPoint's features serve the domain of presentation management. The relatively low value of the CCM may reflect an inherent complexity in the tasks that PowerPoint performs.

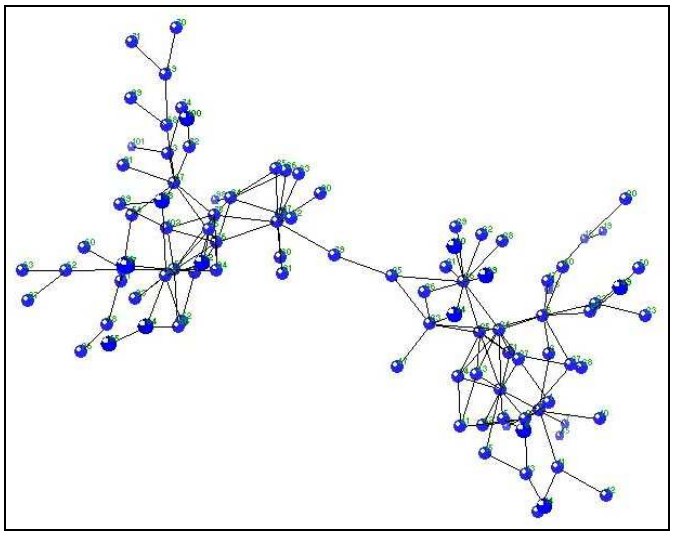


Figure 47 – An artificially constructed graph with a CCM of 10.01

We do know that the CCM is not directly related to the number of concepts even though our data shows an inverse relationship between the number of concepts in an ontology and the CCM. Because the CCM is based on average distance, a graph can be constructed with a low number of nodes and a low CCM. The graph in Figure 47 has 102 nodes organized into two distinct groups. It was created by taking two copies of a recovered ontologies and connecting them. If this graph represented an application ontology, it would have a CCM of 10.01, lower than any of the ontologies constructed in our case studies, including our 1686 node ontology for PowerPoint 2000 (CCM = 14.56). While this counterexample is artificial, it is reasonable to believe that such an ontology could exist, resulting from combining two loosely related groups of concepts under a single morphology.

10.3 Conceptual Coherence and Conceptual Integrity

Conceptual coherence is a first approximation for the conceptual integrity of a computing application. If the ontology lacks coherence, then the application will likely

lack conceptual integrity. The conceptual coherence measures and variation testing techniques that we have developed can be used to guide the development of computing applications. When used with the ontological coverage metrics from use case silhouetting, these methodologies offer developers a means of evaluating their designs, testing for features possessing concepts that will reduce the conceptual coherence of an application. In such cases, a use case silhouette can then be used to measure conceptual fitness with respect to the targeted use context. In the absence of evidence suggesting that this peripheral feature will improve overall usefulness, developers can choose to focus their energies on alternatives. Understanding how to engineer conceptual coherence into these ontologies will bring us closer to ensuring that applications possess a high conceptual integrity.

10.4 Summary of Research Contributions

- A theory of software ontology and *conceptual coherence* as a first approximation of *conceptual integrity*.
- A methodology for *ontological excavation* – the black box reverse engineering of a software ontology – to identify those concepts encoded into the system that are visible and accessible to users of the system.
- Domain-independent analysis methods for identifying the *core concepts* of an application – those concepts that are *essential* to the definition and function of that application.
- A *conceptual coherence metric* applied to the ontology as a first approximation of conceptual integrity.
- A *variation testing* method for distinguishing core concepts from peripheral ones in an ontology.
- Methodology for measuring the ontological coverage of a set of use cases using *use case silhouetting*.
- A method for mapping usability data to the morphology and ontology of an application as an approximation of conceptual fitness.

11 FUTURE WORK

In this dissertation, we have developed theories and methodologies for understanding the conceptual integrity of computing applications and illustrated how conceptual integrity can be related to metrics that measure the potential usefulness of an application relative to a specific use context. While we lack sufficient empirical evidence to state definitively that conceptual coherence directlys measure conceptual integrity or that conceptual integrity and usefulness are directly related, we have prepared the foundations for future exploration in this area. At the very least, our research artifacts offer a means for bridging the gap between the development. In this section, we present several avenues of research that are suggested by our findings.

11.1 Extending the Conceptual Integrity Metric: Conceptual Complexity

In this work, we have shown how conceptual coherence can serve as a first approximation for conceptual integrity. However, coherence is only one aspect of integrity. Intuitively, if a developer used coherence as the only guideline for designing a system and reorganized the ontology to minimize the distance between concepts, they would produce an ontology without peripheral concepts, or reorganize peripheral concepts to strengthen their ties to the core concepts. However, the metrics could also imply that the following reorganization be performed (Figure 48): take all the nodes and attach them to one node to maximize the CCM.

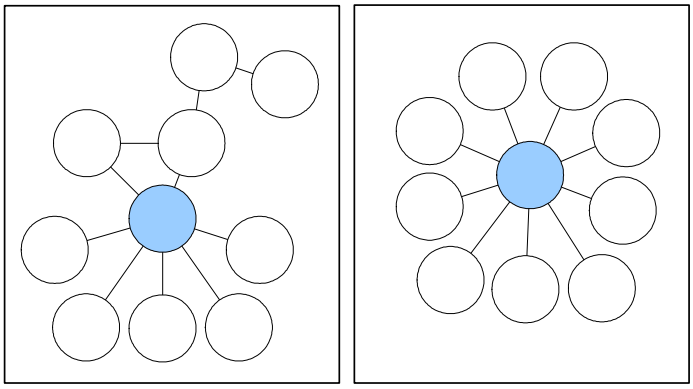


Figure 48 – Improving the conceptual coherence of a graph

Transforming an arbitrary graph with N nodes into a single node with (N-1) degrees would increase the CCM. The graph on the left in Figure 48 has a CCM of 46.38. The graph on the right has a CCM of 55.56. However, in addition to being a difficult heuristic to implement in practice, redistributing the nodes in such a way increases the *conceptual complexity* of the ontology. If conceptual integrity measures how well an ontology's concepts relate to one another, conceptual complexity measures how difficult the concepts are to understand. A complex concept has many attributes, subtypes, and associations. An ontology with many complex concepts will have many interdependencies and a greater chance of producing unintended feature interactions. This ontology will also produce an application that can be difficult for a user to learn or use. We believe that average *degree centrality* can be used to approximate the conceptual complexity of an ontology. Degree centrality in an undirected graph counts the number of edges on a node and compares it to the total number of edges in a graph [194]. We also believe that the degree centrality of individual nodes can be used to highlight concepts that may require extra attention in designing its associated morphological elements or functional organization. We also believe that there may be limits to how complex a

concept can be before it begins to affect the structure of an ontology in an adverse manner.

Designing an ontology to increase coherence while limiting complexity seems to be a difficult task. We are currently working on developing a combined measure of coherence and complexity to develop an enhanced metric of conceptual integrity.

11.2 Studying Conceptual Evolution of Ontologies

Using our ontologies, we can track the conceptual evolution of an application and detect conceptual complexities within the application. Using these same methods, we can also study the conceptual variations amongst a set of applications designed to accomplish the same goals. Identifying precisely what has changed from one version to the next or what is different from one application to the next allows us to study both diachronic and synchronic feature variations and to develop feature taxonomies and patterns similar to Alexander's pattern languages for architectural design [2]. These examples of feature and pattern variations can be used as a basis for designing ontologies specific to use contexts.

11.2.1 Diachronic Variation

In Chapter 9, we presented our large-scale study of the feature evolution of Microsoft Word. What we did not investigate in depth was how existing features varied over time. For example, attributes may migrate from one concept to another over successive versions of the application. In the CD Player example, a Disc has the attribute Artist, which is simply a string for the artist's name. In more modern media players, Artist has become an attribute of the media file or CD track, rather than the entire CD. It migrated from one concept to another, indicating that a data dependency existed between Artist and Track, which can be shown by adopting this representation. Attributes may

also be enhanced with their own attributes, eventually becoming entity types. Continuing the example of Artist, later media players may retrieve information about artists from the Internet, such as biographical data or a recording history of other albums. This additional information makes Artist a concept. Understanding how concepts evolve and develop over time may suggest methods for predicting how features will evolve in a use context over an application's lifetime.

11.2.2 Synchronic Variation

Several applications can possess the same concepts but with different implementations. For example, in our studies, Word and PowerPoint both have Text as a core concept. As an isolated concept, Text is modeled the same way – as a set of characters and special characters. However, in the word processor, Text is contained by paragraphs. In a presentation, Text is contained by slide objects and, specifically, Text Boxes. In a spreadsheet, Text is contained by cells. In a drawing application, such as Visio, Text is contained by Text Boxes contained within drawing objects. All of these use the Text concept but the containing and organizing concepts fundamentally alter how the user interacts with text in those applications. A study of the synchronic variation of similar concepts would reveal different design solutions to related problems. These solutions could seed a library that could be used for generating ontologies.

11.3 Ontological Structures and Software Architecture

From our case studies of ontological excavation, we have observed some distinct characteristics in the recovered ontologies, such as the CD Player's central cluster of core concepts surrounded by attributes, the Protocol Calculator / Calendar's multiple subgroups within its ontology, and Notepad's multiple teleons. We also noticed a

possible correlation between these characteristics and the conceptual coherence metrics for their respective applications. These observations suggest that applications may be categorized by their ontological structures. These structures exhibit common forms that likely emerge from design and evolution. We propose three archetypical ontological structures: the Reef Structure, the Toolbox Structure, and the Urban Structure.

11.3.1 The Reef Structure

A Reef structure represents a system that implements a tightly related set of concepts, possessing a high conceptual coherence. Many metaphors exist that could express the idea of a unified set of concepts constructed around a central architecture. We chose a biological one to account for the evolutionary behaviors that we have observed in the ontologies of single purpose applications. Coral reefs are ecosystems built on a skeleton of calcium deposits created by tiny creatures called coral. This skeleton provides a habitat for many species that each play a role in maintaining the reef environment. Over time, the reef can grow and develop a very rich and stable biological system [12]. A Reef ontological structure has a central structure that not only supports itself but also a number of other entity types that contribute to the overall system.

A Reef-like computing application has an endoskeleton of core services and layers of peripheral services that are supported by the endoskeleton. For example, a spell-checker for a word processor is a peripheral service. Spell checkers could not exist independently of changes for text that has to be spelled correctly. Figure 49 shows an abstract ontology for what we would expect to find in a Reef Structure application – a skeleton of core concepts with some supporting ones on the periphery.

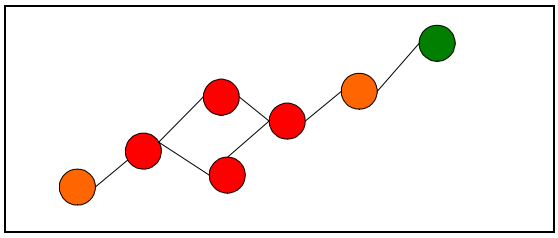


Figure 49 – Reef Structure of Conceptual Coherence

Simply stated, this application does one basic thing. It may have services that aid in the achievement of that central goal but these services could be removed without much loss to the overall application. Arguably, most applications begin as reefs – tools built for a single purpose. Over time, applications with well-defined goals or constrained domains may evolve by acquiring new features but they only tend to acquire those features that can directly support and enhance the existing ones. Most small applications and games – like CD Players, Hearts, Calendars, and Clocks – have Reef ontological structures. We believe that the PowerPoint centrality visualization in Figure 37 exemplifies what we would expect to find in a Reef structure – a central endoskeleton of core concepts surrounded by numerous supporting ones.

11.3.2 The Toolbox Structure

A toolbox that one might find in a home is a collection of tools that have different affordances for specialized tasks. Hammers, screwdrivers, and pliers all contribute to different sorts of tasks, but one usually does not use every tool in a toolbox to accomplish a task. A Toolbox structure (Figure 50) has a collection of conceptually unrelated and lightly related ontologies that have been assembled for reasons of convenience or design under a single morphology.

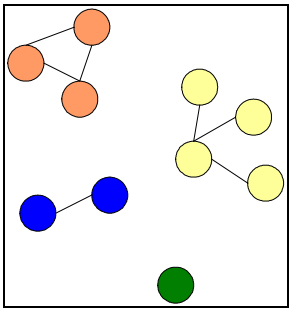


Figure 50 – Toolbox Structure of Conceptual Coherence

The tools in a toolbox collectively support a broader category of goals such as resource management, information management, or media playing. Over time, a Toolbox may collect more tools, enhance the capabilities of its existing tools, or begin merging the tools by combining their functions. Examples of Toolbox model workpiece computing applications include RealOne Player (a media player that supports CD playing, CD burning, Internet radio, web browsing, and MP3 management) and Yahoo! Instant Messenger (ostensibly a instant messaging tool that also delivers information such as weather, stocks, auctions, and news). The overall Toolbox structure will have less conceptual coherence by definition but will be structurally coherent within each individual tool. An example of a Toolbox ontological structure can be found in Appendix 4, which is a case study of the Protocol Calculator / Calendar device.

11.3.3 The Urban Structure

Urban areas are often divided into large neighborhoods that compete for influence, income, resources, business, and desirable populations of people. Sometimes neighborhoods will fragment into smaller zones. Other times, neighborhoods will subsume other less successful neighborhoods. The collective urban environment may be easy to identify on a map but the subordinate areas within that region may not be.

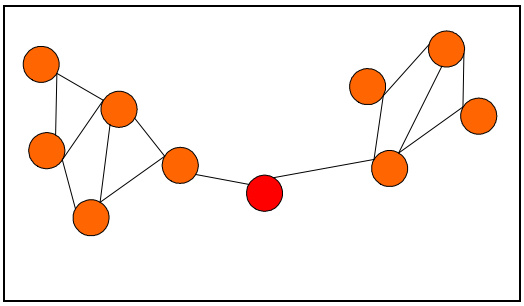


Figure 51 – The Urban Structure of Conceptual Coherence

The Urban Structure (Figure 51) results when an application has acquired features that cause its ontology to lose conceptual coherence. Large clusters of features may merge, blurring the boundaries between some services, or fracture, causing others to become more isolated and independent of the computing application. We expect the ontology of an Urban application to contain competing clusters of core concepts (multiple and unrelated teleons that have similar sizes and influences on the ontology). It differs from the Toolbox Structure in that these core concepts are connected to each other. A Toolbox can have large independent clusters of ontologies because, in practice, each cluster represents a tool that is used independently.

The Urban Structure can result from a design that had poorly articulated or confused requirements. It also might have begun as a Reef or Toolbox model but over time had evolved by growing in size and functionality to acquire new customers that have different and occasionally conflicting, requirements for this application. Over its lifetime, such an application may be perceived to be more bloated by users who find themselves using smaller and smaller percentages of the overall system with each release. Examples of Urban workpiece computing applications include Microsoft Word and Microsoft Excel.

An example of an Urban ontological structure can be found in Appendix 5 which describes the Microsoft Windows Notepad application.

11.3.4 Ontological Structures and Computing Applications

While still hypothetical, these archetypical ontological structures have interesting implications for aiding designers in creating and enhancing their applications. If this and future research show a correlation between usefulness and conceptual coherence, then one could imagine design heuristics encouraging adoption of the Reef or Toolbox ontological structures as a framework for organizing domain concepts in the ontology of an application prior to developing its software architecture. Bloat could be prevented in the ontology by detecting Urban ontological structures in the ontologies of an application version before proposed features are added. In an application with a close correspondence between its concepts and the underlying software architecture, software maintenance activities could include *ontological grafting* and *pruning*: adding teleons to the ontology or removing them to preserve conceptual coherence. For example, if a computing application is found to have an Urban ontological structure, one could preserve the stability and enhance the maintainability of the application by pruning one of the competing clusters of core concepts and creating a separate application that contributes services without sharing morphologies. Using ontological structures to guide application development could represent tremendous payoffs in improving usefulness and in reducing unnecessary development costs.

11.4 Engineering for Conceptual Fitness in a Computing Ecosystem

Developing an application to ensure its conceptual fitness to a use context must also take into account the evolving concerns and goals of the users and organizations that inhabit that use context. We believe successful designs must adapt an organic approach to understanding both fitness and adaptation. We use the metaphor of the computing ecosystem to describe how these applications can be engineered to ensure that conceptual fitness persists over time [94].

11.4.1 The Computing Ecosystem

In biology, ecosystems describe a defined envelope of physical, chemical, and biological processes within a space and time [132]. More generally, an ecosystem is "a system of interacting species in a particular environment" [115]. We formally define a computing ecosystem as *a set of use contexts that use computing to fulfill goals, contained within an environment of interest.* A computing ecosystem can be a single person and a handheld PDA or a multi-national company of database management specialists. In a computing ecosystem, the organisms are the computing functions that users apply towards achieving their goals [94].

The *biological fitness* of an organism is described as "the ability of an individual to produce viable offspring and contribute to future generations" [132]. In principle, the fitness of an individual organism takes secondary importance to overall genetic fitness of the species, as measured by its population size and, in evolutionary biology, by how many years that species managed to survive over the lifetime of the Earth [53, 54]. But this genetic fitness is really a property that emerges from the individual organism's abilities to survive and procreate, summed over the entire population of the species. Thus,

one cannot understand biological fitness solely by studying the genetic code. One must examine the resulting phenotypic expressions – the *physiological features* of an organism expressed by its genetic code. Analyzing the fitness of a species requires studying not only the individual organism's physiological attributes that enable it to reproduce, the most direct contribution to fitness, but how its features enable it to interact successfully with its environment and its fellow organisms in activities like its ability to gather and consume nutrients, escape predation, and maintain homeostasis across climatic variations.

Likewise, for computing applications, code and architecture do not reveal anything about their fitness in a computing ecosystem. The correct unit of study has to be the phenotypic expressions of the software or its features. If an application lacks the features that would make it useful to its users, then it lacks sufficient fitness for it to exist in the computing ecosystem. We now need a more precise characterization of the relationship between an application's features and concepts and the computing ecosystem.

11.4.2 The Use Niche and Feature Fitness

Biologists tend to think of the habitats of particular organisms in a more narrow context – that of the *ecological niche*. An ecological niche is a physical environment that supplies the food and space required for the survival of a set of species [132]. Species that occupy the same ecological niche will compete for these resources. Over time, only the species that have evolved or adapted a sufficient level of fitness will survive in these niches.

Features in computing applications inhabit *use niches*, a bounded space in the computing ecosystem that contains subsets of the ecosystem's resources and goals. A use niche may be as broad as "document writing" and as narrow as "database sorting". The

fitness of a feature in a use niche is primarily determined by that feature's ability to fulfill the requirements of that niche. However, a feature can possess the necessary concepts and functions to occupy a niche but still be unfit due to poor usability, which we characterize as a conflict between the available energy in the ecosystem and the usage cost of the feature.

11.4.3 Use Potentials and Activation Cost

In a use niche, the resources that allow features to exist can be collectively abstracted to what we call *use potentials*. A use potential is the amount of available energy or effort that the users are willing to expend to activate the features of the application. On the application side, features have an *activation cost* that represents the corresponding amount of energy or effort required to engage those services. For example, a GOMS (Goals, Operations, Methods, Selection) formulation, from research in human-computer interaction, measures this activation cost in terms of the number of user interface item selections [42]. If a feature can achieve the goals of a niche and its activation cost is less than the use potential of the niche it occupies, then it has a high potential fitness.
For example, a user could write a document, such as a memo, in a spreadsheet or a word processor. The word processor has a low activation cost for its text processing functions because it has been designed that way. The spreadsheet application has some of the same concepts related to text but has a higher activation cost because its ontology has been designed around the management and analysis of numerical data. For a normal memo, it would make sense to choose the word processor. However, if a user wanted to write a memo with charts and graphs displaying financial information, it may be easier to use the

spreadsheet application as the word processor will have a higher activation cost for those features.

11.4.4 Usefulness and Usability

We have shown through use case silhouettes that conceptual coherence can be used to estimate the probable usefulness of a computing application. Software developers have approached this problem from the perspective of improving software quality through testing activities and formal methods, user interface design and usability engineering, applying empirical methods to requirements development and end-user evaluations to measure perceived usefulness.

Researchers and developers have recognized that the requirements guiding the design and implementation of the eventual system have to be written from a thorough understanding of the user's domain. This understanding can be best obtained by interacting with actual users in their working environment [52, 165, 175, 178, 191]. A number of empirical techniques have been developed to derive user requirements directly from the use context. These include incorporating ethnographic methods [80, 105], contextual design [26, 27, 97], intent-based specifications [136], and inquiry-based analysis [177]. In the human-machine systems area, ecological interface design and ecological task analysis use empirical studies of the work domain to improve the design of user interfaces [70, 122, 205, 206].

11.4.5 Engineering Fitness Into Applications

Evolving computing applications so that their fitness improves over time without a corresponding increase in complexity. We now argue, using a thought experiment, how the computing ecosystem framework and its subordinate concepts – use niches, use

potentials, and activation cost – can characterize both system fitness and how that fitness can decrease or increase over time.

A thought experiment for understanding use niches is to imagine a suite of desktop productivity tools (e.g. Microsoft Office), not as a collection of programs, but as a collection of functions unbundled from their arbitrary boundaries. Instead of a word processor, a database tool, a spreadsheet application, and so on, there are simply a collection of functions for processing text, managing graphics, saving files, copying objects, sorting data, and so on. Now imagine some environment that uses these tools, like an accounting office or academic department. Over a year, we could collect data on the use of these functions and eventually we would have a characteristic profile that could represent the fitness of all of the productivity suite's features relative to that computing ecosystem.

With a detailed analysis of the computing ecosystem, one might discover unused use potentials and untapped niches. For example, prior to Microsoft PowerPoint, people developed presentations by handwriting them on slides or printing slides using some word processor. This showed that a use niche existed for applying computing to the problem of presentation development with a high use potential but few existing features that could take advantage of this. When PowerPoint was released, it supplied features that displayed a high fitness for the use niche of presentation creation that the word processor's features could no longer occupy that niche.

Thus, with a hypothetical profile of an idealized application with high conceptual fitness coupled with knowledge of potential features that will be required in the future, one could imagine reengineering applications to contain only those features that have

some fitness in the ecosystem to reduce perceptions of bloat and to reduce unnecessary complexity in the system. We can imagine that if someone wanted their product to remain competitive against similar applications, they would want to engineer the architecture to support features that may be required in the future or to reduce the activation cost of frequently used features to improve that application's fitness.

12 CONCLUSION

In this dissertation, we have developed a research framework, methods, and models that allow us to analyze and measure the conceptual integrity of an application using its conceptual coherence as a first approximation. We have shown how our techniques for identifying core concepts and measuring conceptual coherence can be obtained structurally and verified empirically. We have provided examples of our methodology and models taken from actual working systems and validated some of them from external sources.

Conceptual integrity ultimately derives from the an artifact's design quality, insofar as the design concerns conceptual, structural, and functional relationships. Unlike design activities in architecture and the fine arts, conceptual integrity does not directly measure any sense of aesthetic, which is, appropriately, a difficult property to understand and nearly impossible to quantify. Yet, Brooks's use of architectural aesthetics in the edifice of a cathedral to explain conceptual integrity has some interesting parallels with our discussions of conceptual integrity in computing applications. In the computing world, a word occasionally used to describe extraordinary applications or tools is "elegance". Computing professionals appreciate elegance, whether at the algorithm level, like Dijkstra's shortest-path algorithm [188], or at the system level, like the Linux kernel [181]. Something possessing elegance exhibits high levels of computational power, requiring only minimal effort to use.

In a sense, pejoratives like "bloat" and "feature creep" reflect a perception that an application lacks this elegance. This is somewhat ironic because such applications typically have a large and varied set of features that grant them tremendous capabilities,

while modern user interfaces and operating systems allow users to access these features in complete ignorance of the technical aspects of their implementation. But, ultimately, elegance and usefulness are in the eye of the beholder. “Bloat” and “feature creep” are really ways of saying “this application does more than what I need and the unused features are getting in my way.”

Brooks says the following, in his seminal essay, “No Silver Bullet – Essence and Accident in Software Engineering”:

> “The essence of a software entity is a construct of interlocking concepts: data sets, relationships among data items, algorithms, and invocations of functions. This essence is abstract, in that the conceptual construct is the same under many different representations. It is nonetheless highly precise and richly detailed.
>
> I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it and testing the fidelity of the representation. [34]”

The difficulties in designing the morphologies and ontologies of computing application are primarily conceptual. Unless the system has been designed to implement purely mechanical or analytical functions, such as those in many embedded systems, design quality, as measured by an application’s usefulness, usability, and conceptual integrity, depends entirely on the designer’s abilities to comprehend the domain of the use context and to transform that comprehension into requirements and specifications. Even with extensive expertise and experience in such activities, the lack of instrumentation or theories that can supply feedback on design as it pertains to the application’s intended purpose reduces the design process to activities of guesswork. While designs can be enhanced by iterative feedback from the users of the system or by using prototypes that

are iteratively adapted until they are deemed satisfactory, much effort could be saved with a methodological approach to developing the ontology of the application.

We have designed our methods to be usable on any computing application, or, possibly, any information artifact, provided that the user of the application and the morphology have been framed properly. Our method of silhouetting, for measuring the conceptual fitness of an application to a use context, can be applied to any application through use cases, scenarios, user actions, or qualitative user data. While the black box nature of these techniques and the current lack of instrumentation make them somewhat labor-intensive, the time spent performing these analyses is still trivial when weighed against the maintenance costs of reworking a system that has been delivered and rejected by its users. In the future, our results will serve as a foundation for developing complimentary methods that will enable us to measure an application's conceptual integrity, encompassing all of its functional and computational aspects. When this has been achieved, we will have not only identified tools to enable developers to achieve Brooks's prescription of engineering conceptual integrity into the system but will have taken the state of computing application development one step closer to maturity.

APPENDIX A – GLOSSARY

This is a glossary of special terminology used in this dissertation. Underlined words denote terms that were coined or re-defined for this specific research.

activation cost	The amount of effort required by a user to access a service provided the application
actor	A type of user of a computing system. [105]
adaptation	The process of changing attributes and behaviors of something to better suit a specific context. In biology, it also describes a physiological attribute of a species that improves its fitness relative to an element of the surrounding ecosystem. [53, 54]
aggregation	A whole/part relationship where one class of entity types represents a larger thing which consists of smaller things. Denoted by a 'has-a' relationship. [26]. In our modeling conventions, we break the traditional convention of requiring both things to have independent identities in the case of attributes. However, attributes themselves are not permitted to have has-a relationships.
association	A structural relationship that specifies that elements of one type are connected to elements (concepts) of another type. [26]
attribute	An intrinsic property of a thing in the real world [192]. In our model of an application ontology, an attribute is a concept that lacks independent existence except as a property of an entity type.
betweenness centrality	A prestige measure that measures the number of geodesics between all pairs of nodes in the graph that use a particular node. The higher the centrality measure, the more other nodes depend on that node.

	Because leaf nodes only serve as start and end points for paths, they automatically have a betweenness value of 0. [194]
bipartite graph	A graph in which the nodes can be partitioned into two subsets such that edges always connected nodes taken from the different subsets. Bipartite graphs are used to model two-mode networks. [194]
black box reverse engineering	The recovery of some computing application domain model, behavior, or attribute without reference to the code used to implement that computing application.
bloat	The term used to describe a computing application possessing a disproportionate number of unnecessary services that interfere with the normal or desired use of this application.
closeness centrality	A prestige measure that measures the average distance from a subject node to all other nodes. [194]
computing application	Any device or system that uses some form of computation to accomplish a goal. Also the term that can refer to ‘application’, ‘computing artifact’, ‘software’, ‘software application’, and ‘software system’.
computing ecosystem	A set of use contexts that use computing to fulfill goals, contained within an environment of interest [94].
concept	A generalized idea of a thing or class of things. [177] In our model of an application ontology, either an entity type or an attribute can be a concept.
conceptual coherence	A property of a computing application measuring the degree to which the concepts contained within its ontology are tightly related.
conceptual complexity	A property of a computing application measuring the

	degree to which the concepts contained within its ontology are easy to understand.
conceptual fitness	The property of a computing application that assesses how well its ontology matches the domain of the use context in which it is being used.
conceptual integrity	The property of a system designed under a unified and coordinated set of design ideas. [34]
container	A morphological element that contains and structures interactors [96]
core concept	A concept that is essential to defining a computing application's feature set and identity. [96]
correction	A software maintenance activity that applies repairs to errors in the code [157]
customer	The purchaser of the computing application. Not necessarily the user of the application.
degree centrality	A prestige measure that uses the number of edges on a node (its degree). A value of 1.0 on a scale of 0.0 to 1.0 means the node has edges leading to all other nodes in the graph. [194]
density	The number of edges in a graph divided by the possible number of edges. Also called network density. [29]
diachronic variation	Variation across time – usually in reference to evolution or development.
digraph	A directed graph. [194]
display	A morphological element that makes both static and dynamic data about the computing application's states available to the user. [96]
domain model	"A definition of the entities, operations, events, and relationships that abstract commonalities or regularities in a domain, together with a classification

	of these.” [8]
ecosystem	“An ecosystem is a system of interacting species in a particular environment.” [115]. Defines a system of interest where the granularity could be the object of study (like a species) or a set of arbitrary conditions. [132]
eigenvector centrality	A prestige measure that measures the centrality of a node relative to the importance of its surrounding nodes. [194]
enhancement	A software maintenance activity that adds new features, generally visible to the users of the system. [157]
entity	A “thing” that can be distinctly identified. [45]
entity type	A set of entities that have the same attributes. [59]
E-type program	A software system that solves a problem or implements a computer application in the real world. [120]
evolution	The process of change over a period of time. In the biological sense, evolution refers to the physiological changes that a species experiences through the process of mutation, natural selection, and reproduction. [53, 54]
feature	A user-accessible behavior or service implemented by a computing application.
feature aggregation	An evolutionary behavior of a computing application where it acquires new features at every stage of release.
feature creep or creeping featurism	The “tendency to add to the number of features that a device can do, often extending the number beyond all reason.” [154]
fitness	Attribute of an entity that assesses its ability to inhabit

	a specific context. In biology, fitness describes the ability of an organism or a species to survive long enough to reproduce.
generalization	A relationship between a kind of entity type (parent or superclass) and a more specific kind of that entity type (type, child, or subclass). Denoted by an “is-a” relationship. [26]
geodesic	The shortest path between a pair of nodes. [194]
improvement	A software maintenance activity that applies an optimization to performance, usability, maintenance, or other nonfunctional properties of a computing application. [157]
instance	A concrete manifestation of an entity type [26].
information centrality	A prestige measure that measures the information contained in all paths originating with a specific node. [194]
interactor	A morphological element that can be directly accessed or manipulated by the user of a system. [96]
k-core	A connected, maximal, induced subgraph of nodes such that each node has a minimum degree greater than equal to *k* [63].
morphological element	A component that forms the structure of a computing application’s morphology.
morphological map	A graph modeling the elements that compose the morphology of a computing application and their relationships to each other. [96]
morphology	The external presentation of a computing application consisting of those elements that are both user accessible and perceivable.
niche	A place and functions that a species has in an ecosystem

ontological construction	The process of modeling excavated concepts and relationships as a semantic network.
ontological coverage	A metric that measures the proportion of the ontology covered by a set of concepts.
ontological excavation	The process of using black-box reverse engineering to recover a computing application's ontology. [96]
ontological grafting	The process of adding concepts or a set of concepts and their relationships to an ontology.
ontological pruning	The process of removing concepts or a set of concepts and their relationships from an ontology.
ontological structure	A structural pattern in the ontology that organizes the concepts and their relationships.
ontology	A representation of set of concepts used for domain or data modeling. [32, 65, 78, 140, 192, 193]. Also the study of being – of existence and its relationship to nonexistence [115].
operations	The activities that a system performs.
perceived ease of use	The degree to which a person believes that using a particular system is free of effort [52]
perceived usefulness	The degree to which a person believes that a particular system could enhance his or her job performance [52]
peripheral concept	A concept which is considered optional to an application's definition. [96]
portal	A mapping from the morphology of a computing application to a concept or set of concepts in the ontology. [95]
prestige measure	A prestige measure assesses the importance of a node relative to the rest of a graph. [194]
problem domain	A collection of items of real-world information that have the following characteristics: 1) "deep or comprehensive relationships among the items of

	information are suspected or postulated with respect to some class of problems" and 2) the problems are perceived as significant by the members of the community. [8]
problem frame	A diagram that describes the class, characteristics of the problem domain, and a central concern for a class of problems. [102]
reef ontological structure	An ontological structure with a core that exists not only to support itself but also a number of other entity types that contribute to the overall system.
relationship	A reference or association that exists between entity types. [59]
requirement	A description of how a system should behave or a description of a system property or attribute. [179]
semantic network	The collection of all the relationships that concepts have to other concepts [180]. Semantic networks are the first ontology models to make use of graphical formalism and were developed as psychological models of human memory [16] [177]. Generically speaking, they are graphical representations of a body of facts [152].
service	A service is an operation or series of operations performed by an application that performs a task for a user.
social network	A graph or network that encapsulates people or social groups and their relationships to one another. [194]
software evolution	The process of adapting a computing application or system during the software maintenance phase of its development. Also the description of the changes that software experiences over its lifetime.
software product family	"A set of products that share architectural properties,

	features, code, components, middleware, or requirements." [131]
software product line	"A set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way." [109]
subtype	A *subtype* is a specialization of an entity type [26].
superpositioned graph (supergraph)	A graph containing a computing application's morphological map, the ontology, and the interconnections that link a morphological element (portal) to the concepts that it reveals. Used to derive the bipartite graph of morphological elements and concepts.
synchronic variation	The variation of features across different entities of the same type within the same time frame.
teleon	An identifiable substructure of an ontology that suggests features at the user level. Consists of a set of concepts that have strong interrelationships. [95]
toolbox ontological structure	An ontological structure consisting of conceptually unrelated and lightly related ontologies that have been assembled for reasons of convenience or design under a single morphology.
urban ontological structure	An ontological structure with multiple clusters of core concepts that are loosely connected to each other.
usability	An attribute of an application that measures how much effort is required to activate an affordance or service provided by that application.
use case	"A use case specifies a sequence of actions, including variants, that the system can perform and that yields an observable result of value to a particular actor."

	[105]
use case coverage	A metric for the proportion of concepts in an ontology covered by a set of use cases.
use case silhouette	The set of concepts that have been activated or illuminated by a set of use cases or a sequence of morphological element activations.
use context	A use context consists of the external physical (or virtual) environment that contains the computing application and its users, the goals that the combined computing application/user system wishes to achieve, and the various factors (business rules, customer demand, user and system capabilities) that govern the operation and performance of both the environment and the completion of those goals
use niche	A bounded space in the computing ecosystem that contain subsets of the ecosystem's resources and goals [94].
use ontology	The use ontology consists of only those concepts that are actually used in a specific use context.
use potential	A use potential is the amount of available energy or effort that the users are willing to expend to activate the features of the application [94].
usefulness	The extent to which an application succeeds in assisting a set of users to achieve a set of goals, relative to the amount of effort required to engage those features
user	The person, group of people, or entity that uses a computing application.
workpieces problem	"A *problem* of developing a tool to support creation and editing of text or other machine-readable objects." [102]

APPENDIX B – LIST OF SOFTWARE APPLICATIONS USED FOR THIS DISSERTATION

- Microsoft Excel 2000 – Spreadsheet application used to organize data and calculate statistics.
- Microsoft Visio 2002 Professional – Drawing application used to create the morphological maps and ontology diagrams.
- UCINET 6.0 – Social Network Analysis tool used to perform the computations for ontological analysis and conceptual integrity [29].
- Net Draw 1.1 – Social Network drawing tool used to calculate and visualize *k*-cores and to produce the data for our 3D visualizations [27].
- Mage 6.36 – 3D Visualization tool used to visualize our graphs [170].

APPENDIX C – SOURCE CODE FOR MICROSOFT VISIO MACRO

This source code was originally written by David Yu under the supervision of Dr. Melody Moore at Georgia State University. The code was modified by Idris Hsi to produce a lookup file with no numbers.

It takes a Microsoft Visio diagram as input and generates three files:

- DL.txt – An adjacency list style representation of a graph (Visio drawing objects and connectors) present in the Visio diagram. This implementation generates an undirected graph and can be modified to generate directed graphs. The DL format is one of several standard notations used to represent graphs [29].
- Lookup.txt – A lookup file providing an index of labels with the associated node numbers in the DL.txt file.
- Lookup_no.txt – A lookup file with a list of labels without the associated node numbers for easier copying to other documents.

It uses the following Microsoft Visual Basic Libraries (accessible through Visual Basic → Tools → References)

- Visual Basic For Applications
- Microsoft Visio 2002 Type Library (Service Release 1)
- OLE Automation
- Microsoft Forms 2.0 Object Library
- Microsoft Scripting Runtime.

Known Issues:

- *Hard Coded File Paths* – The file paths are currently hard coded to a pre-specified location for convenience.
- *Dropping Labels* – The macro looks at an object's text to write it to file. However, if an object has not been labeled properly, specifically when a set of

objects grouped to form one object has not been designated to select the group rather than the member shapes and a member shape has been given the label, the macro will not write the text to the file. However, this side effect was useful for identifying problems with stencil shapes in a Visio diagram.

- *One Dimensional vs. 2 Dimensional objects* – The macro automatically ignores 1-D objects, such as lines or shapes configured in Visio to behave like lines.

Source Code:

```
Sub ShowPageConnections()
 Dim objFileSystem As FileSystemObject
 Dim myText As TextStream
 Dim lookUp As TextStream

 Set objFileSystem = CreateObject("Scripting.FileSystemObject")
 Set myText = objFileSystem.CreateTextFile("C:\Temp\DL.txt")
 Set lookUp = objFileSystem.CreateTextFile("C:\Temp\Lookup.txt")
 Set lookUp_nonum = objFileSystem.CreateTextFile("C:\Temp\Lookup_no.txt")

 'Pages collection of document
 Dim pagsObj As Visio.Pages
 'Page to work on
 Dim pagObj As Visio.Page
 'Object From connection connects to
 Dim fromObj As Visio.Shape
 'Object To connection connects to
 Dim toObj As Visio.Shape
 'Connects collection
 Dim consObj As Visio.Connects
 'Connect object from collection
 Dim conObj As Visio.Connect
 'Type of From connection
 Dim fromData As Integer
 'String to hold description of From connection
```

```
Dim fromStr As String
'Type of To connection
Dim toData As Integer
'String to hold description of To connection
Dim toStr As String
Dim conshapes As Shapes
Dim conshape As Shape
Dim finalshape As Shape

Dim connects2 As Connects
Dim connect2 As Connect
Dim shape2 As Shape

Dim string1 As String
Dim string2 As String

Dim count As Integer
Dim tempcount As Integer
count = 1
'assuming 9999 is the maximum number of objects
Dim shapeArray(9999) As Integer

'Get the Pages collection for the document
'Note the use of ThisDocument to refer to the current document
Set pagsObj = ThisDocument.Pages
'Get a reference to the first page of the collection
Set pagObj = pagsObj(1)
'Get the Connects collection for the page
Set conshapes = pagObj.Shapes
'Debug.Print conshapes.Count

'populate the lookup file
For Each conshape In conshapes
If Not conshape.OneD Then
lookUp.Write count & " " & conshape.Text
shapeArray(conshape.ID) = count
lookUp.WriteBlankLines (1)
```

```
count = count + 1
End If
Next

'populate the lookup file
For Each conshape In conshapes
If Not conshape.OneD Then
lookUp_nonum.Write conshape.Text
lookUp_nonum.WriteBlankLines (1)
End If
Next

'populate the dl file's header section
myText.WriteLine "dl n=" & count - 1
myText.WriteLine "format = nodelist1"
myText.WriteLine "labels:"

For tempcount = 1 To count - 1
myText.Write tempcount & " "
Next

myText.WriteBlankLines (1)
myText.WriteLine "Labels embedded"
myText.WriteLine "Data:"

!traverse the shapes in the Visio diagram
For Each conshape In conshapes

If Not conshape.OneD Then
myText.Write shapeArray(conshape.ID) & " "


Set consObj = conshape.FromConnects
For Each conObj In consObj
Set finalshape = conObj.FromSheet
'myText.Write finalshape.Text & " "
```

```
Set connects2 = finalshape.Connects
For Each connect2 In connects2
Set shape2 = connect2.ToSheet
string1 = shape2.ID
string2 = conshape.ID
If StrComp(string1, string2) Then
 myText.Write shapeArray(shape2.ID) & " "
End If

Set shape2 = Nothing
Next

Set finalshape = Nothing
Next

myText.WriteBlankLines (1)
End If

Next

End Sub
```

Sample Input:

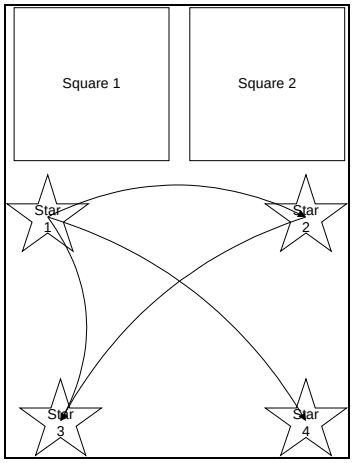


Figure 52 – Sample Visio 2002 diagram

DL.TXT File (from Sample Diagram - see Figure 52)

An adjacency list showing number of nodes, type of format, labels used for the nodes, and the adjacency list. Numbers are used in place of text labels to improve readability of the data formats in the other tools. In the example below, nodes 1 and 2 are isolates while node 3 is connected to nodes 4, 5, and 6.

```
dl n=6
format = nodelist1
labels:
1 2 3 4 5 6
Labels embedded
Data:
1
2
3 4 6 5
4 3 5
5 3 4
6 3
```

LOOKUP.TXT File (from Sample Input - see Figure 52)

A lookup table with the labels from the Visio diagram.

1 Square 1
2 Square 2
3 Star 1
4 Star 2
5 Star 3
6 Star 4

LOOKUP_NO.TXT file (from Sample Input - see Figure 52)

A lookup table with the labels from the Visio diagram without index numbers for easier copying into spreadsheets.

Square 1
Square 2
Star 1
Star 2
Star 3
Star 4

APPENDIX D – LEGEND FOR MORPHOLOGICAL AND ONTOLOGICAL DIAGRAMS

These are the symbols and abbreviations used in the morphology (Table 32) and ontology diagrams.

Table 32 – Morphological Map Symbols and Abbreviations for Elements

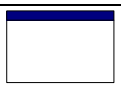
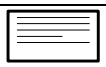
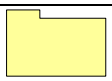
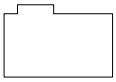
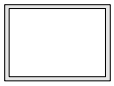
Name	Abbr.	Symbol	Type	Description
Application	A	 <Application>	Container	Application (can also refer to an external application accessed by the application being studied)
Button	B	 <Button> B	Interactor	Button or something with button functionality (like a selectable icon)
Check Box	CB	 <Check Box> CB	Interactor	A Check Box selector
Dialog Box	DB	 <Dialog Box> DB	Container	Any generic dialog box
Dialog Box (File Handling)	DBFH	 <Dialog Box> DBFH	Container	A dialog box specifically for file handling (opening and saving files)
Dialog Box (Prompt)	DBP	 <Dialog Box> DBP	Container	A user prompt
Display	D	 <Display> D	Display	Any generic display of information or data – not interactive.
Drop-down List	DD	 <Drop-Down Field> DD	Container	A drop-down list of items
Folder	F	 <Folder> F	Container	A file folder
Hyperlink	H	 <Link to> H	Interactor	A hyperlinked item or selector
Interactive Display	ID	 <Interactive Display>	Interactive Object	A selectable display or one that contains interactive elements. Often used in the context of games.
Information Field	IF	 <Information Field> DI	Display	An non-interactive information field that displays data or values (like date and time)
Interactive Object	IO	 <Interactive Object> IO	Interactive Object	An object that can be moved and selected (like a drawing object), an embedded or linked object, (e.g. from another application), or a virtual object (e.g., a game object).

Table 32 Continued

Name	Abbr.	Symbol	Type	Description
Interactive World	IW	 <Interactive World> IW	Interactive Object	A virtual world, usually in the context of a game.
Keyboard	K	 <Action> K	Interactor	Can refer to the keyboard or a set of keystrokes. Usually used for highly interactive programs like games. Otherwise assumed to exist.
Keyboard Shortcut	KS	K <Keyboard Shortcut> KS	Interactor	A hotkey or command line
List	L	 <List> L	Container	Any list of selectable items, usually found in a dialog box. Usually a scrolling list.
List Item	LI	<selection item> LI	Interactor	A selectable item in a list.
Main Window	MW	 <Window> MW	Container	The main window of the application
Menu	M	<Menu Item> M	Container	A menu, usually found off of a menu bar
Menu Bar	MB	File Edit ... ... <Menu> MB	Container	A menu bar of an application
Menu (Hanging)	MH	File Edit ... ... <Hanging Menu>	Container	A hanging (sometimes tear-away) menu. Sometimes accessed by a right mouse button click on an interactive object.
Menu Item	MI	<Menu Item> MI	Interactor	A selectable item in a menu that brings up a dialog box.
Menu Item (Action)	MIA	<Menu Item> MI	Interactor	A menu item that performs an operation in the application (e.g. Undo, Copy, Cut, Save, Exit)
Menu Item (Check Box)	MIC	<Menu Item Check Box> MIC	Interactor	A menu item with a check box used for setting options in the application.
Menu Item (Drop Down)	MID	<Menu Item Drop Down> MID	Interactor	A menu item with a drop down list or sub-menu.

Table 32 Continued

Name	Abbr.	Symbol	Type	Description
Mouse Action	MsA	 <Mouse> MsA	Interactor	A mouse action – dragging or pointing. Usually used to model specialized interactions, such as drawing or game playing. Otherwise assumed to be part of the application.
Mouse Left / Right Button	MsLB / MsRB	 <Mouse> MsLB  <Mouse> MsRB	Interactor	An action using a specific mouse button – usually in the context of playing a game or accessing a special menu with the right mouse button.
Radio Button	RB	 <Radio Button> RB	Interactor	A radio button selector
Selector	S	<selection item>	Interactor	Any generic selector that does not fit in this scheme (e.g. a color palette selector in a drawing tool)
Slide Selector	SS	 <Slide Selector> SS	Interactor	A selector with a sliding handle to choose from a range of values
Table	T	 <Table> T	Container	A table with lines, columns, and cells
Text Field	TF	 <Text Field> TF	Interactor	Any field or area where text can be entered.
Toolbar	TB	 <Toolbar> TB	Container	A toolbar of buttons and other selectors
Tabbed Pane	TP	 <Tabbed Pane> TP	Container	A tabbed pane, usually found in a dialog box with multiple modes.
Web Page	WP	 <Web Page> WP	Container	Any web page (usually viewed through an external browser, assuming the application being studied is not a web browser).
Window	W	 <Window> W	Container	A generic container referring to a viewable portion of an application or dialog box. Also contains interactive elements.
Work Artifact	WA	 <Work Artifact> WA	Interactive Object	In work product systems, the artifact being produced (e.g. a document, a drawing, a presentation, a spreadsheet).

Morphological Element Notation

- **Top Level Element** – An element directly accessible by the user – usually a container, display, or an interactive object

  Notation: `<name> <abbreviation>`

Example: PowerPoint Main Window → `PowerPoint MW`

Example: Embedded Microsoft Table → `Microsoft Table IO`

- **Contained Element** – An element found in a container – usually an interactor or display.

  Notation: `<container name> <container abbreviation> : <name> <element abbreviation>`

  Example: 'Find' Dialog Box – 'Find What' Text Field →

  `Find DB: Find What TF`

  Example: 'Clock Window' – 'Time' Display → `Clock W: Time IF`

- **Inferred or Specified Name** – A name not taken directly from the morphology but inferred by the modeler to identify an morphological element or to make a generic label more specific.

  Notation: `[<element name>]`

  Example: the Web Page 'Save As' File Handling Dialog Box →

  `Save As [Web Page] FHDB`

Table 33 – Ontology Symbols and Description

Name	Symbol	Type	Description
Aggregation	— has-a ▶	Relationship	Indicates a part/whole relationship between two entity types. Arrow is drawn from the containing entity type (whole) to the entity type (part). Also used for attributes (part) of an entity type (whole).
Association (line)	——▶	Relationship	Indicates a structural relationship with another concept. Arrow is drawn from the entity type to the association diamond.
Association	<Relationship>	Relationship	Specifies the structural relationship that between two or more entity types. The relationship is described as a verb. Arrows drawn from an entity type to the relationship indicate the entity type initiating the relationship. Arrows drawn from the relationship to an entity type indicate the object of this relationship.
Attribute	<Attribute>	Concept	A property of an entity type.
Entity Type	<Entity Name>	Concept	Defines a set of entities that have the same attributes.
Generalization (Is-A)	— is-a ▶	Relationship	A relationship between a kind of entity type (parent) and a more specific kind of that entity type (child). The arrow is drawn from child to parent.

Inferred or Specified Concept Name – A name not taken directly from the morphology but inferred by the modeler to qualify a concept or specify a concept more thoroughly.

Notation: `[<entity type or attribute name>]`

Example: the Presentation currently being edited by the user →

`[Active] Presentation`

Example: Black and White check box selector that allows a Presentation to be viewed in Black and White → `Black and White [Setting]`

APPENDIX E – SAMPLE KINEMAGE FILE FOR 3D VISUALIZATIONS

A Kinemage is a 3-D structure displayable by the visualization program Mage [170]. A Kinemage file produced using NetDraw [27] or UCINET [29] has the following sections:

- Header – Contains information about the general configuration of the Kinemage.
- Node List – The node list, attributes, and coordinates for displaying the nodes as balls in the Kinemage.
- Label List – The labels and their attributes associated with the nodes and their coordinates.
- Edge List – The edges, their attributes, and their start and end coordinates.

Using the CD Player as an example, we can generate the following Kinemage (Figure 53):

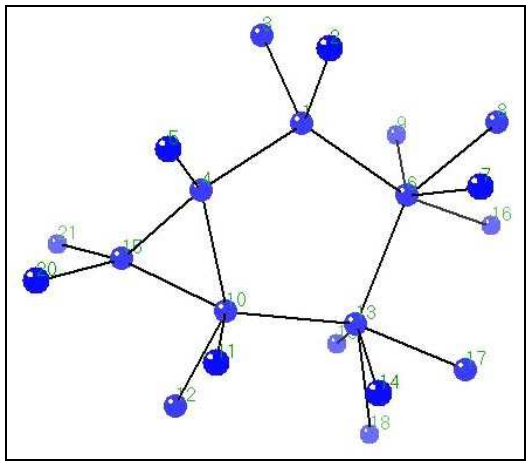


Figure 53 – Visualization of the CD Player Ontology

Below are samples from each section of the Kinemage file generated by NetDraw.

Header

@kinemage 0

@caption

@fontsizelabel 20

@zoom 1.00

@thickline

@zclipoff

@group {FromNetDraw.kin} animate movieview=1

Node List

@subgroup {nodes} dominant

@balllist 0 color=blue radius=0.150

{1 1} 4.8636 6.5490 6.8139

{2 2} 5.2956 7.8966 8.2081

{3 3} 4.3189 8.0404 5.6342

{4 4} 2.9784 5.4942 7.3623

{5 5} 2.3203 6.3187 8.9903

Label List

@subgroup {labels} dominant

@labellist 0 color=green

{1} 4.8636 6.5490 6.8139

{2} 5.2956 7.8966 8.2081

{3} 4.3189 8.0404 5.6342

{4} 2.9784 5.4942 7.3623

{5} 2.3203 6.3187 8.9903

Edge List

@subgroup {ties} dominant

@vectorlist {ties} color=black radius=0.250 width=2 angle=20

P 4.8636 6.5490 6.8139

5.2956 7.8966 8.2081

P 4.8636 6.5490 6.8139

4.3189 8.0404 5.6342

P 4.8636 6.5490 6.8139

2.9784 5.4942 7.3623

APPENDIX F – LIST OF ONTOLOGICAL COMPONENTS IN POWERPOINT 2000 ONTOLOGY

The following table contains a list of the sub-ontologies modeled to produce the PowerPoint 2000 ontology.

Table 34 – Sub-ontologies of PowerPoint 2000

Name	Description
[Configuration]	Inferred concept. The options and settings for the main application.
[Current_Slide] Notes	Inferred concept. The notes associated with the current slide.
[Find_Replace_Tool]	Inferred concept. The Find and Replace text feature of PowerPoint.
[Graphics_File_Type]	Inferred concept. Any file type that stores images.
[Output_Device]	Inferred concept. The device that will be projecting the slide show.
[PowerPoint]_File	Inferred concept. Any work artifact savable by PowerPoint (web page, presentation, slide show).
[PowerPoint]_Table	PowerPoint's Table (composed of multiple text boxes).
[Print_Job]	Inferred concept. A print job with print settings that is sent to a printer to be printer.
[Printer]_Document_Properties	The document property settings of a printer.
[Slide Image]	Inferred concept.
[Slide_Index_Display]	Inferred concept – displays all the slide images of the active presentation.
[Slide_Master_Image]	Inferred concept – The image of the Slide Master.
[Sound_Recorder]	Inferred Name - Tool that records a sound from a microphone to play back during a presentation.
[Style_Checker_Tool]	Inferred concept – Checks the text in a presentation for conformance to style settings set by the user.
[Text_Animation]	Inferred concept – An animation specific to text.
[View]	Inferred concept – A generic view setting.
3D_Direction	The direction of a 3D effect on a draw object.
3D_Lighting	The lighting of a 3D effect on a draw object.
3D_Setting	The setting of a 3D effect on a draw object.
3D_Surface	The direction of a 3D effect on a draw object.
Action_Item	An item in Meeting Minutes that can be sent to Microsoft Outlook.
Action_Setting	A property of a Slide Object that determines what an object does when selected during a Slide Show.
Active_Presentation	The current presentation being edited or displayed.
Address_Book	An address book of contacts that can be invited to an Online Broadcast.
Alignment	An alignment of text relative to the border of a text box.

Table 34 Continued

Name	Description
Animation	A generic animation of a slide object.
Animation_Preview	A tool for previewing an animation setting.
AutoContent_Wizard	A Wizard that helps the user generate a presentation skeleton.
AutoCorrect_Exception	A setting for the AutoCorrect tool that contains the strings to ignore.
AutoCorrect_Tool	The tool responsible for automatically correcting specific strings of text.
AutoLayout_Type	A slide layout applied to a slide.
Autoshape [Draw] Object	An AutoShape drawing object.
AutoShape_Action_Button	An AutoShape Action Button – has action settings.
AutoShape_Basic_Shape	An AutoShape Basic Shape.
AutoShape_Block_Arrow	An AutoShape Block Arrow.
AutoShape_Callout	An AutoShape Callout.
AutoShape_Connector	An AutoShape Connector – actually a type of Line – connects shapes together.
AutoShape_Flowchart	An AutoShape Flowchart shape.
AutoShape_Line	An AutoShape Line – actually a type of Line.
AutoShape_Stars_and_Banners	An AutoShape Stars and Banners shape.
Background	The background colors and images for a slide or handout.
Black_and_White_[Setting]	A View setting that displays the slides in grayscale.
Border	A line used to outline a drawing object.
Border_Setting	The color and width settings for borders.
Broadcast	The event that displays an online presentation.
Bullet	A character used in front of a line to highlight it.
Bulleted_List	A list of lines preceded by a bullet.
Case_Style	Text can have a case style – e.g. UPPERCASE or Title Case.
Category	A property of Microsoft Chart.
CD_Audio_Track_[Sound]	A track on a CD.
Cell_Alignment	The alignment of text contained in a table cell.
Change_Case_[Tool]	A tool that allows the user to change the case of a selection.
Character	An individual character in a string.
Chart_Effects	Animations that highlight items in a chart.
ClipArt	A graphics object obtained from the Clip Art Gallery.
ClipArt_Gallery	A library of graphics objects organized by category.
Color	A color.
Color_Image_Control_Setting	A color setting for picture objects.
Color_Scheme	Color settings for a presentation organized by type.
Column_Text	An object in a slide that structures text into multiple columns.
Comment	An object that behaves like a Post-It note in a presentation.
Contact	The information for a person that can be invited to an online presentation.
Current_Handout	The current handout being modified.
Current_Notes_Page	The current notes page being modified by the user.
Current_Slide	The current slide being modified by the user.
Custom_Property	A custom property of a presentation.
Custom_Show	A customized sequence of slides in a presentation.
Date_Area	The area of the slide that displays the date.

Table 34 Continued

Name	Description
Design_Template	A template that applies preset color schemes and graphics to a presentation.
Dictionary	A list of correctly spelled words referenced by the Spelling Tool.
Draw_Table_Tool	A tool that enables a user to draw a table.
Drawing_Tool	A tool that allows a user to draw graphics in a presentation.
Embedded_Object	A object generated external to PowerPoint and that can be inserted into and displayed in a presentation.
End_Punctuation_Style	A style setting that determines how a sentence should be punctuated.
Entry_Animation	Animation associated with an object as it enters a slide show during a presentation.
File	An electronic data format stored on a computer.
Fill	A shading used to fill a shape or the background of an object.
Find_People_[Tool]	Tool in online broadcast to locate participants in database.
Font_Format	The display settings for a character, including typeface, color, point size, and style.
Footer	The text displayed in the Footer Area.
Footer_Area	A text box that contains footer text in a master.
Genigraphics_Contact_[Info]	Contact information for the Genigraphics Wizard.
Genigraphics_Creative_Support_and_Service	A service provided by Genigraphics.
Genigraphics_Order	A description of services, contact information, and payment information that is sent to Genigraphics along with a presentation.
Genigraphics_Payment_Method	The method chosen by the user for paying Genigraphics.
Genigraphics_Presentation_Material_[Product]	Types of materials provided by Genigraphics for printing slides.
Genigraphics_Wizard	The wizard provided by the Genigraphics company for submitting presentations to be processed.
GIF_Player	A window that plays animated GIFs.
Gradient	A continuum blend of two different colors.
Group	A set of draw objects linked as a single object.
Handout	A page that displays slide images and/or notes to hand out to the audience.
Handout_Background	The background settings for a handout.
Handout_Master	The master settings for a handout.
Header	The text displayed in the Header Area.
Header_Area	A text box that contains header text in a master.
Header_Footer_Setting	A setting for headers and footers.
Hyperlink	A setting for text or an object that specifies a destination that can be navigated to during a slide show by clicking on it.
Hyperlink_destination	A location that can be hyperlinked.
Line	Any line in PowerPoint.
Line_Spacing	The number of points between lines of text.
Link	An active (automatically updating) or passive link to an external object.
Macro	A script that performs a series of actions.
Master_Text_Style	The master font formats for levels of body text in a slide.

Table 34 Continued

Name	Description
Media_Clip	A clip object that plays either an animation, movie, or sound.
Media_Player	A tool that plays a media clip.
Meeting	An event where a group of people assemble to discuss a particular purpose.
Meeting_Minder_[Tool]	A tool that runs parallel to a Slide Show to take minutes and action items for a meeting.
Microsoft_Chart	An external object that displays data in a graphical format.
Microsoft_Word_[Write-Up]	The format of a presentation sent to Microsoft Word.
Motion_[Clip]	A clip art object that displays an animation.
Movie	A file that stores a movie.
Movie_Object	An object that plays a movie during a slide show.
Multimedia_[Animation]_Setting	A setting for movie and sound objects.
Narration	A recording that accompanies a presentation.
Narration_Sound_Quality	A setting for the quality of a recording.
Normal_View	The standard three paned view of PowerPoint displaying the outline, notes, and current slide.
Notes	Text associated with a particular slide.
Notes_Background	The background images and fill for a Notes Page.
Notes_Body_Area	The text box on a Notes Pages that contains the main body of notes.
Notes_Master	The formatting styles, layout, and backgrounds that structure all Notes Pages for a presentation.
Notes_Page	The page associated with a slide that displays a slide image and notes associated with that particular slide.
Notes_Page_Object	Any object that can be inserted into a notes page.
Notes_Page_View	The view in the application that displays the notes page.
Number_Area	A text box in a master that shows the slide number.
Numbered_List	A list of text where each line is preceded by a number.
Numbered_List_[Number]	A number that precedes a line of text in a numbered list.
Object_Area	A text box in a layout that contains text and lists of text.
Online_Broadcast_[Tool]	A tool for setting up the online broadcast of a presentation.
Outline	A list of all the slide titles in a presentation.
Outline_View	The view in the application that displays only the outline.
Outlining_Tool	A tool that assists the user in developing and organizing an outline.
Pack_and_Go_Wizard	A wizard that helps the user package a presentation with a slide show viewer for displaying on a different computer.
Page_Number_[Field]	A field that automatically updates the page number.
Page_Setup	A set of parameters that format a page to be printed.
Paragraph	A string of characters that terminates in a carriage return.
Pattern	A graphic pattern that used in a fill.
Picture	An image file that can be inserted into a presentation.
Picture_[Clip]	A clip art image.
Picture_[Object]	An object that displays an image.
Presentation	A collection of slides and notes pages.
Presentation_Broadcast_Lobby_Page	The web page generated by the Online Broadcast tool where attendees log in to view the meeting.
Presentation_Options	Presentation attributes that determine whether to display the title, footer, slide number, and date last updated. Used in the AutoContent Wizard.

Table 34 Continued

Name	Description
Presentation_Statistics	Statistics, such as number of words, associated with a Presentation
Presentation_Style	Style settings for a presentation.
Presentation_Type	A type of presentation as described by the AutoContent Wizard.
Presentation_Web_Page	A web page generated to display a presentation.
Preset_Animation	An animation setting preset by the developers.
Printer	A device that prints print jobs.
Projector_Wizard	A wizard that assists the user in connecting a computer to a projector.
Publish_As_Web_Page_Tool	A tool that allows a user to publish a presentation as a web page.
Recolor_Chart_[Tool]	A tool that assists a user in changing the colors used in a chart.
Recolor_Picture_[Tool]	A tool that assists a user in changing the colors used in a picture.
Record[ed]_Sound	A sound recorded from microphone.
Replace_Font_[Tool]	A tool that globally replaces one font typeface with another.
Reviewing_Tool	A tool for reviewing comments inserted into a presentation.
Selection	A highlighted region containing text or objects selected by the mouse pointer.
Send To [Destination]	A destination for sending the presentation.
Server_Options	Settings for the server managing the online broadcast.
Shadow	A shading displayed below an object.
Show_Type	A method for displaying a slide show.
Slide Number [Field]	A text box that displays a slide number.
Slide	A page in a presentation that displays images and text in a slide show.
Slide_Finder	A tool for finding a slide in a presentation.
Slide_Layout	A setting that structures the slide text boxes and objects in a specific manner.
Slide_Master	A master setting for all slides in a presentation.
Slide_Miniature	A small window that displays what the slide will look like during a slide show.
Slide_Navigation_Controls	A set of controls that are embedded on a web page generated from a presentation.
Slide_Number_[Field]	A field that automatically updates the slide number.
Slide_Object	Any object that can be placed on a slide.
Slide_Page_View	The view displayed by the application showing only the slide.
Slide_Show	A sequence of slides displayed to an output device.
Slide_Sorter_View	The view displayed by the application showing the images of all slides in an application.
Slide_Transition	An animation in a slide show that occurs between slides.
Slide_View	A mode in PowerPoint that displays the slide.
Sound	The output of an audio file.
Sound_[Clip]	A clip art object that plays a sound.
Sound_Object	An object that plays a sound.
Special_Character	A character without a displayable token that has special properties (e.g. tab, space, carriage return).
Spelling_[Tool]	A tool that checks spelling in the document.

Table 34 Continued

Name	Description
Subtitle_Area	The area of slide that contains the subtitle.
Subtitle_Text	The text used in a subtitle.
Summary_Slide	A slide generated by PowerPoint that contains a summary of the presentation.
Symbol	A specially displayed character.
Table	A grid structure that contains objects (such as text) in its individual cells.
Table_Cell	A box created by the intersection of a column and row in a table.
Table_Eraser	A tool used to erase cell borders to merge or delete them in a table.
Text	A string of characters and special characters.
Text_Box	A box that contains text and can be contained by a drawing object.
Text_Box_[Cell]	A text box that belongs to a table.
Time_and_Date_[Field]	An automatically updating field that displays the time and date.
Title_Area	The area of a slide that contains the title.
Title_Master	The master settings for the title slide of a presentation or template.
Title_Slide	The first slide in a presentation.
Title_Text	Text used in a title.
Word	A string of text bordered by special characters (space, tab, paragraph mark).
WordArt	Refers to object that structures text with a set of graphics and colors in a particular shape.
WordArt_Shape	Structure and outline of WordArt object.
WordArt_Text	Text used to create WordArt object.

APPENDIX G – USE CASES FROM POWERPOINT 2000 FOR WINDOWS FOR DUMMIES

This is a list of use cases used to produce the use case silhouette in Chapter 7.5.

Table 35 – List of Use Cases from *PowerPoint for Dummies for Windows* [129]

Chpt #	Use Case Name (Chapter or Section Title)	Description
1	Starting PowerPoint 2000	Starting PowerPoint, Opening a Presentation
1	Help Me, Mr. Wizard?	Using the AutoContent Wizard
1	Which Way Is Up	Survey of screen elements
1	The View from Here is Great	Overview of views
1	Zooming In	Using the zoom feature
1	Editing Text	Editing text
1	Moving from Slide to Slide	Navigating between slides
1	Fifty Ways to Add a New Slide	Adding different types of slides to a presentation
1	Outline That For Me!	Using the outline mode
1	Printing That Puppy	Printing a presentation or handout
1	Saving Your Work	Saving a presentation
1	Retrieving a Presentation from a Disk	Opening a presentation
1	Closing a Presentation	Closing a presentation
2	Working with Objects	Description and use of slide objects
2	Selecting Objects	Selecting slide objects
2	Resizing or moving an object	Resizing or moving slide objects on a slide
2	Editing a Text Object: The Baby Word Processor	Editing text in text objects
2	Using the arrow keys	Using arrow keys
2	Moving around faster	Extra keyboard commands for navigating around a slide
2	Deleting text	Deleting text
2	Marking Text for Surgery	Selecting text
2	Using Cut, Copy, and Paste	Using the Clipboard
2	Cutting or copying a text block	Cutting and copying a block of text to the Clipboard
2	Pasting text	Pasting text from the Clipboard
2	Cutting, copying, and pasting entire objects	Selection operations with slide objects
2	Deleting a Slide	Deleting slides from a presentation
2	Duplicating a Slide	Copying a slide within a presentation.
2	Finding Text	Finding text in a presentation
2	Replacing Text	Replacing text in a presentation
2	Rearranging Your Slides in Slide Sorter View	Using the slide sorter
3	Switching to Outline View	Using outline view
3	Understanding Outline View	Components of the outline view

Table 35 Continued

Chpt #	Use Case Name (Chapter or Section Title)	Description
3	Selecting and Editing an Entire Slide	Selecting a Slide
3	Selecting and Editing One Paragraph	Selecting a paragraph
3	Promoting paragraphs	Moving paragraphs one level up in an outline
3	Demoting paragraphs	Moving paragraphs one level down in an outline
3	Dragging paragraphs to new levels	Moving a paragraph to an arbitrary level
3	Adding a New Paragraph	Starting a new paragraph in the Outline View
3	Adding a New Slide	Adding a new slide in the Outline View
3	Moving text the old-fashioned way	Moving text up or down in the outline view using keystrokes
3	Moving text the dragon drop way	Moving text up or down in the outline view using the mouse
3	Collapsing an entire presentation	Collapsing the presentation in outline view
3	Expanding an entire presentation	Expanding the presentation in outline view
3	Collapsing a single slide	Collapsing a slide's paragraphs
3	Expanding a single slide	Expanding a slide's paragraphs
3	Showing and Hiding Formats	Viewing formats in outline mode
3	Creating a Summary Slide	Creating a Summary Slide
3	Busting Up a Humongous Slide and Using the Presentation Assistant	Reducing the number of bullet points using the Office Assistant's help
4	The Over-the-Shoulder Spell Checker	Using the spell checker while typing
4	After-the-Fact Spell Checking	Using the spell checker directly
4	Capitalizing correctly	Using the Change Case Tool to capitalize text properly
4	Using Style Checker Options	Using the style checker
5	Understanding the Notes Page View	Using the Notes Page View
5	Adding Notes to a Slide	Adding Notes to a slide
5	Adding an Extra Notes Page for a Slide	A trick to create a second notes page for a particular slide
5	Adding a New Slide from Notes Page View	Adding a new slide from the Notes Page View
5	Printing Notes Pages	Printing a Notes Page
5	Random Thoughts about Speaker Notes	How to use Speaker Notes for a presentation
6	The Quick Way to Print	Printing overview
6	Using the Print Dialog Box	Changing print options
6	Changing printers	Changing printers
6	Printing part of a presentation	Printing selected parts of a presentation
6	Printing more than one copy	Printing multiple copies of a presentation
6	What do you want to print?	Selecting the item to print

Table 35 Continued

Chpt #	Use Case Name (Chapter or Section Title)	Description
6	What are all those other checkboxes?	Additional print options
8	Changing the Look of Your Characters	Formatting text
8	To boldly go	Changing the font style of text to Bold
8	Italics	Changing the font style of text to Italics
8	Underlines	Adding underlines to text
8	Big and little characters	Creating super and subscript text
8	Text fonts	Changing text font (typeface)
8	The color purple	Changing text color
8	The shadow knows	Adding shadows to text
8	Embossed text	Creating embossed text
8	Biting the Bullet	Adding bullets to text
8	Centering text	Center justifying text
8	Flush to the left	Left justifying text
8	Flush to the right	Right justifying text
8	Stand up, sit down, justify!	Justifying text
8	Messing with Tabs and Indents	Setting tabs and indents
8	Spacing Things Out	Adjusting line spacing
9	Working with Masters	Using the Master slide templates
9	Changing the Slide Master	Editing the Slide Master
9	Adding recurring text	Adding text to the Slide Master
9	Changing the Master color scheme	Change the color scheme of the presentation
9	Changing the Title Master	Editing the Title Master
9	Changing the Handout Master	Editing the Handout Master
9	Changing the Notes Master	Editing the Notes Master
9	Overriding the Master text style	Changing text format on individual slide
9	Changing the background for just one slide	Changing background for an individual slide
9	Adding a date, number or footer to slides	Adding a date, number, or footer to a slide from the dialog box
9	Adding a header or footer to Notes or Handout pages	Adding a header or footer to Notes or Handout page from the dialog box
9	Editing the header and footer placeholders directly	Editing header and footer directly in the Master
9	Applying a different template	Changing the template used to format the Presentation
9	Creating a new template	Creating a new presentation template
9	Creating a new default template	Setting a template to be the default template
10	Using Color Schemes	Description of presentation color schemes
10	Using a different color scheme	Changing the current color scheme to another

Table 35 Continued

Chpt #	Use Case Name (Chapter or Section Title)	Description
10	Overriding the color scheme	Customizing specific objects with a different color scheme
10	Changing colors in a color scheme	Modifying a color scheme
10	Shading the slide background	Shading the background
10	Using other background effects	Using the gradient, pattern, or picture effects for a background
10	Applying color to text	Changing the color of text
10	Changing an object's fill or line color	Changing the color of a fill or line
10	Creating a semi-transparent object	Setting a color to be semi-transparent
10	Copying color from an existing object	Using Format Painter to copy an existing color
11	Free pictures!	Using the Clip Art Gallery
11	Dropping In Some Clip Art	Adding Clip Art to a presentation
11	Moving, Sizing, and Stretching Clip Art	Changing clip art settings
11	Boxing, Shading, and Shadowing a Picture	Using pictures
11	Editing a Clip Art Picture	Editing a clip art picture
11	Colorizing a Clip Art Picture	Changing a clip art colors
11	Getting Clip Art from the Internet	Downloading clip art to the presentation
11	Inserting Pictures without Using Clip Gallery	Alternative methods for adding pictures
12	Zoom In	Using the zoom feature
12	Display the ruler	Using the ruler
12	Stick to the color scheme	Managing color formatting
12	Save frequently	Saving a presentation
12	The Drawing Toolbar	Review of the drawing toolbar features
12	Drawing Simple Text Objects	Drawing an object
12	Drawing straight lines	Drawing a line
12	Drawing rectangles, squares, ovals, and circles	Drawing specific AutoShapes
12	Using AutoShapes	Using the AutoShape feature
12	Drawing a Polygon or Freeform Shape	Drawing a unspecified shape
12	Drawing a Curved Line or Shape	Drawing a curved line or shape
12	Setting the Fill, Line, and Font color	Setting the color of a Fill, Line or Font
12	Setting the Line Style	Changing the style of a line
12	Applying a Shadow	Adding a shadow to a draw object
12	Adding 3-D Effects	Adding 3-D effects to draw objects
12	Flipping an object	Changing an object’s orientation
12	Rotating an object 90 degrees	Changing an object’s rotation
12	Using the Free Rotate button	Using the Free Rotate tool
12	Changing layers	Moving a draw object between layers
12	Line 'em up	Aligning draw objects
12	Using the guides	Using the guides to place objects
12	Group therapy	Grouping draw objects together
13	Adding Charts and Graphs to PowerPoint Slides	Inserting Charts and Graph objects

Table 35 Continued

Chpt #	Use Case Name (Chapter or Section Title)	Description
13	Creating a Chart	Creating a chart object
13	Inserting a new slide with a chart	Inserting a new slide with a chart object
13	Inserting a chart in an existing slide	Inserting a chart object into an existing slide
13	Moving and Resizing a Chart	Changing a chart's dimensions
14	Creating an Organizational Chart	Creating an Organizational Chart object in the Microsoft Organizational Chart application
14	Inserting a new slide with an organizational chart	Adding a new slide with an Organizational Chart object
14	Inserting an organizational chart in an existing slide	Adding an Organizational Chart to a slide
15	Using WordArt	WordArt functionality overview
15	The PowerPoint 2000 Better Table Maker	Adding a table from PowerPoint
15	Inserting a table from Microsoft Word	Adding a Microsoft Word Table
16	All about sound files	Managing sound objects in a presentation
16	Inserting a sound in PowerPoint	Adding a sound object
16	Playing an embedded sound	Playing a sound object during a presentation
16	Removing a sound	Removing a sound object
16	Using transition sounds	Using sounds during slide transitions
16	Adding a movie to a slide	Adding a movie object to a slide
16	Playing a movie	Playing a movie object
17	Slide transitions the easy way	Adding a slide transition in the Slide Sorter view
17	Slide transitions the other way	Adding a slide transition using the Slide Show menu
17	Text animation the easy way	Adding text animation in the Slide Sorter view
17	Text animation the other way	Adding text animation using the Slide Show menu
17	Animating Other Slide Objects	Animating slide objects
17	Using the Predefined Animation Effects	Using Preset Animations
17	Setting Up a Presentation That Runs by Itself	Setting up a slide show that automatically forwards to the next slide
18	Using Hyperlinks	Adding hyperlinks to slides
18	Creating a hyperlink to another slide	Adding a hyperlink to another slide
18	Creating a hyperlink to another presentation	Adding a hyperlink to another presentation
18	Removing a hyperlink	Removing a hyperlink from an object
18	Button actions	Adding action settings to buttons
18	Button shapes	Overview of different action buttons
18	Creating a button	Creating a button in a presentation
18	Creating a navigation toolbar	Creating a navigation toolbar for a web-based presentation

Table 35 Continued

Chpt #	Use Case Name (Chapter or Section Title)	Description
18	Using the Web Toolbar	Overview of Web toolbar functions
19	About the Save as a Web Page Feature	Saving a presentation as a web page
19	Web Page Publishing Options	Setting web page publishing options for a presentation
19	Publishing a Presentation or HTML File to the Web	Publishing a presentation as a web page
20	Using Presentation Broadcast	Setting up an Online Broadcast
20	Scheduling a Broadcast	Scheduling an Online Broadcast
20	Meet Now with NetMeeting	Using NetMeeting
21	Editing More Than One Presentation at a Time	Editing multiple presentations
21	Stealing Slides from Other Presentations	Moving and copying slides between presentations
21	Exploring Document Properties	Presentation document properties
22	Creating a presentation from a foreign file	Importing other file types into PowerPoint to create a presentation
22	Inserting slides from an outline	Inserting slides from a Microsoft Word outline
22	Exporting an Outline	Exporting an outline from a presentation
22	Saving Slides as Graphics Files	Saving slides as a graphics file
23	Using the File --> Open Command	Opening an existing presentation
24	Using a Local Photo Lab	Creating 35 mm slides from PowerPoint
24	Using Genigraphics	Using the Genigraphics Wizard
25	Setting Up a Slide Show	Setting up a slide show of a presentation
25	Starting a Slide Show	Running a slide show
25	Keyboard and mouse tricks for your slide show	Managing a slide show using the keyboard or mouse
25	The John Madden effect	Using the Pen feature during the slide show
25	Rehearsing Your Slide Timings	Using the Rehearse Timings dialog box
25	Using the Pack and Go Wizard	Using the Pack and Go Wizard to compress a presentation and a viewing tool.
25	Loading a packed presentation on another computer	Transferring a packed presentation to another computer
25	Running a slide show by using the Viewer	Using the Viewer to view a packed slide show
25	The Meeting Minder	Using the Meeting Midner
25	Running a Presentation over a Network	Using Online Broadcast
25	Using Custom Shows	Using the Custom Shows feature
25	Creating a custom show	Creating a custom show.
25	Showing a custom show	Selecting a custom show to display.

REFERENCES

[1] C. Alexander, *The Timeless Way of Building*. New York, NY: Oxford University Press, 1979.

[2] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. New York, NY: Oxford University Press, 1977.

[3] N. Anquetil, "Characterizing the informal knowledge contained in systems," presented at Eight Working Conference on Reverse Engineering, Stuttgart, Germany, 2001.

[4] A. I. Antón, M. McCracken, and C. Potts, "Goal Decomposition and Scenario Analysis in Business Process Reengineering," presented at 6th International Conference on Advanced Information System Engineering (CAiSE'94), Utrecht, The Netherlands, 1994.

[5] A. I. Antón and C. Potts, "Functional Paleontology: System Evolution as the User Sees It," presented at 23rd International Conference on Software Engineering, Toronto, Canada, 2001.

[6] A. I. Antón and C. Potts, "Requirements Engineering in the Long-Term: Fifty Years of Telephony Feature Evolution," presented at International Conference on Software Engineering, Toronto, ON, 2001.

[7] M. Aoyama, "Continuous and Discontinuous Software Evolution: Aspects of Software Evolution across Multiple Product Lines," presented at 4th International Workshop on Principles of Software Evolution, Vienna, Australia, 2001.

[8] G. Arango, "Domain Analysis Methods," in *Software Reusability*, W. Schafer, R. Priéto-Diaz, and M. Matsumoto, Eds. Chichester, England: Ellis Horwood, 1994, pp. 17-49.

[9] G. Arango and R. Prieto-Díaz, "Domain Analysis Concepts and Research directions," in *Domain Analysis and Software Systems Modeling*, R. Priéto-Diaz and G. Arango, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1991, pp. 9-26.

[10] L. J. Arthur, *Rapid Evolutionary Development*. New York, NY: John Wiley & Sons, 1992.

[11] L. J. Arthur, *Software Evolution: The Software Maintenance Challenge*. New York, NY: John Wiley and Sons, 1988.

[12] D. Attenborough, *The Living Planet: A Portrait of the Earth*. Boston, MA: Little, Brown and Company, 1985.

[13] R. Baecker, K. Booth, S. Jovicic, J. McGrenere, and G. Moore, "Reducing the Gap Between What Users Know and What They Need to Know," presented at ACM Conference on Universal Usability 2000, 2000.

[14] B. Balzer, "Living With COTS," presented at International Conference on Software Engineering, Orlando, FL, 2002.

[15] R. Barker, *CASE*Method: Entity-Relationship Modelling*. New York, NY: Addison-Wesley, 1990.

[16] A. Barr and E. A. Feigenbaum, "The Handbook of Artificial Intelligence," vol. 1. New York , NY: Addison-Wesley, 1989.

[17] E. Barry, S. Slaughter, and C. F. Kemerer, "An Empirical Analysis of Software Evolution Profiles and Outcomes," presented at 20th International Conference on Information Systems, Charlotte, North Carolina, 1999.
[18] D. Barstow and G. Arango, "Designing Software for Customization and Evolution," 1991.
[19] J. A. Bateman, "Ontology Construction and Natural Language," presented at Workshop on Formal Ontology in Conceptual Analysis and Knowledge Representation, PAdova, 1993.
[20] C. Batini, S. Ceri, and S. B. Navathe, *Conceptual Database Design: An Entity-Relationship Approach*. Reading, MA: The Benjamin/Cummings Publishing Company, Inc., 1992.
[21] D. Batory, "A Tutorial on Feature Oriented Programming and Product Lines," presented at International Conference on Software Engineering, Portland, Oregon, 2003.
[22] D. Batory, C. Johnson, B. Macdonald, and D. Von Heeder, "Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study," *ACM Transactions on Software Engineering and Methodology*, vol. 11, pp. 191-214, 2002.
[23] D. Benyon, T. Green, and D. Bental, *Conceptual Modeling for User Interface Development*. London: Springer-Verlag, 1999.
[24] H. Beyer and K. Holtzblatt, *Contextual Design: Defining Customer-Centered Systems*. San Francisco: Morgan-Kaufmann Publishers, Inc., 1998.
[25] G. Booch, *Object-Oriented Analysis and Design*. Reading, MA: The Benjamin/Cummings Publishing Company, Inc., 1994.
[26] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.
[27] S. P. Borgatti, *Netdraw*. Harvard: Analytic Technologies, 2002.
[28] S. P. Borgatti and M. G. Everett, "Models of Core/Periphery Structures," *Social Networks*, pp. 375-395, 1999.
[29] S. P. Borgatti, M. G. Everett, and L. C. Freeman, *UCINET 5.0 Version 1.00*: Analytic Technologies, 1999.
[30] A. Borgida, R. J. Brachman, D. L. McGuinness, and L. A. Resnick, "CLASSIC: A Structural Data Model for Objects," presented at SIGMOD International Conference on Management of Data, Portland, Oregon, 1989.
[31] T. F. Bowen, F. S. Dworack, C. H. Chow, N. Griffeth, G. E. Herman, and Y.-J. Lin, "The Feature Interaction Problem in Telecommunications Systems," presented at Seventh International Conference on Software Engineering for Telecommunication Switching Systems, Bournemouth, UK, 1989.
[32] R. J. Brachman, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Resnick, "Living with CLASSIC: When and How to Use a KL-ONE-Like Language," in *Principles of Semantic Networks*, J. Sowa, Ed.: Morgan Kaufmann Publishers, 1990.
[33] S. Brand, *How Buildings Learn*. New York, NY: Penguin Books, 1994.
[34] F. Brooks, *The Mythical Man-Month*. Reading, MA: Addison-Wesley, 1995.
[35] K. Brooks, "Dancing with digital interface complexity: a story approach," *IEEE Multimedia*, vol. 9, pp. 8-11, 2002.

[36] M. Bunge, *Ontology I: the Furniture of the World*, vol. 3. New York, NY: D. Reidel Publishing Co., Inc., 1977.
[37] M. Bunge, *Ontology II: A World of Systems*, vol. 4. New York, NY: D. Reidel Publishing Co., Inc., 1979.
[38] E. Burd, S. Bardley, and J. Davey, "Studying the process of software change: an analysis of software evolution," presented at Seventh Working Conference on Reverse Engineering, Brisbane, Qld. Australia, 2000.
[39] E. J. Cameron, N. Griffeth, Y.-J. Lin, M. E. Nilson, W. K. Schnure, and H. Velthuijsen, "Towards a Feature Interaction Benchmark for IN and Beyond," *IEEE Communications Magazine*, vol. 31, pp. 64-69, 1993.
[40] E. J. Cameron and H. Velthuijsen, "Feature Interactions in Telecommunications Systems," *IEEE Communications Magazine*, vol. 31, pp. 18-23, 1993.
[41] D. T. Campbell and J. Stanley, *Experimental and Quasi-Experimental Designs for Research*. Palo Alto, CA: Houghton-Mifflin Co., 1963.
[42] S. K. Card, T. P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum Associates, 1983.
[43] L. Carroll, *Through the Looking-Glass*. London, UK: Anness Publishing Limited, 1999.
[44] E. Chang, E. Gautama, and T. S. Dillon, "Extended Activity Diagrams for Adaptive Workflow Modellling," presented at Fourth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, 2001.
[45] P. P. Chen, "The Entity-Relationship Model - Toward a Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, pp. 9-36, 1976.
[46] E. J. Chikofsky and J. H. Cross, "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, vol. 7, pp. 13-17, 1990.
[47] R. Clayton, S. Rugaber, and L. Wills, "Dowsing: A Tool Framework for Domain-Oriented Browsing of Software Artifacts," presented at 13th IEEE International Conference on Automated software Engineering, 1998.
[48] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. Cambridget, MA: MIT Press, 1994.
[49] M. A. Cusumano and R. W. Selby, *Microsoft Secrets*. New York, NY: The Free Press, 1995.
[50] K. a. E. Czarnecki, Ulrich W., *Generative Programming: Methods, Tools, and Applications*. Boston, MA: Addison-Wesley, 2000.
[51] A. M. Davis, *Software Requirements Analysis and Specification*. Englewood Cliffs, NJ: Prentice-Hall, 1991.
[52] F. D. Davis, "Perceived Usefulness, Perceived Ease of Use, and User Acceptance of Information Technology," *MIS Quarterly*, pp. 318-339, 1989.
[53] R. Dawkins, *The Blind Watchmaker*. New York: W.W. Norton and Company, 1987.
[54] R. Dawkins, *The Selfish Gene*. New York: Oxford University Press, 1989.
[55] R. De Millo, D. Guindi, K. King, W. M. McCracken, and J. Offut, "An Extended Overview of the Mothra Software Testing Environment," presented at Second Workshop on Software Testing, Verification, and Analysis, Banff, Canada, 1988.

[56] J. M. DeBaud, B. Moopen, and S. Rugaber, "Domain Analysis and Reverse Engineering," presented at International Conference on Software Maintenance, Victoria, British Columbia, Canada, 1994.
[57] J.-M. DeBaud and K. Schmid, "A Systematic Approach to Derive the Scope of Software Product Lines," presented at International Conference of Software Engineering, Los Angeles, CA, 1999.
[58] A. Dix, J. Finlay, G. Abowd, and R. Beale, *Human-Computer Interaction*. New York, NY: Prentice-Hall, 1993.
[59] R. Elmasri and S. B. Navathe, *Fundamentals of Database Systems*. New York, NY: Addison-Wesley, 1994.
[60] M. El-Ramly, E. Stroulia, and P. Sorenson, "From Run-time Behavior to Usage Scenarios: An Interaction-Pattern Mining Approach," presented at 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Edmonton, Alberta, Canada, 2002.
[61] M. El-Ramly, E. Stroulia, and P. Sorenson, "Recovering Software Requirements from System-User Interaction Traces," presented at 14th International Conference on Software Engineering and Knowledge Engineering, Ischia, Italy, 2002.
[62] D. W. Embley, B. D. Kurtz, and S. N. Woodfield, *Object-Oriented Systems Analysis: A Model-Driven Approach*. Englewood Cliffs, NJ: Prentice-Hall, 1992.
[63] M. G. Everett and S. P. Borgatti, "Peripheries of Cohesive Subsets," *Social Networks*, pp. 397-407, 1999.
[64] M. W. Eysenck and M. T. Keane, *Cognitive Psychology: A Student's Handbook*. East Sussex, UK: Lawrence Erlbaum Associates Ltd., 1992.
[65] R. d. A. Falbo, G. Guizzardi, and K. C. Duarte, "An Ontological Approach to Domain Engineering," presented at International Conference on Software Engineering and Knowledge Engineering (SEKE'02), Ischia, Italy, 2002.
[66] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Second ed. Reading, MA: Addison-Wesley, 1990.
[67] M. Fowler, *Refactoring*. Reading, MA: Addsion-Wesley, 1999.
[68] D. France!,"Notre-Dame Cathedral, Chartres, France,"From, available at http://www.discoverfrance.net/France/Cathedrals/Chartres/Notre-Dame_Chartres.shtml, 2005.
[69] W. J. Freeman, "The Physiology of Perception," *Scientific American*, vol. 264, pp. 78-85, 1991.
[70] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1994.
[71] W. W. Gibbs, "Taking Computers to Task," in *Scientific American*, vol. 277, 1997, pp. 82-89.
[72] M. W. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study," presented at International Conference on Software Maintenance, San Jose, CA, 2000.
[73] T. R. G. Green and D. R. Benyon, "The Skull Beneath The Skin: Entity-Relationship Models of Information Artefacts," *International Journal of Human-Computer Studies*, vol. 44, pp. 801-828, 1996.
[74] S. J. Greenspan, J. Mylopoulos, and A. Borgida, "Capturing More World Knowledge in the Requirements Specification," in *Domain Analysis and Software*

Systems Modeling, R. Priéto-Diaz and G. Arango, Eds. Los Alamitos, CA: IEEE Computer Society Press, 1991, pp. 53-62.

[75] R. M. Greenwood, B. Warboys, R. Harrison, and P. Henderson, "An Empirical Study of the Evolution of a Software System," presented at 13th IEEE Conference on Automated Software Engineering, Honolulu, HI, 1998.

[76] M. L. Griss, J. Favaro, and M. d'Alessandro, "Integrating Feature Modeling with the RSEB," presented at Fifth International Conference on Software Reuse, 1998.

[77] T. R. Gruber, "Ontolingua: A Mechanism to Support Portable Ontologies," Stanford University, Technical Report June 1992 1992.

[78] T. R. Gruber, "Toward Principles for the Design of Ontologies Used for Knowledge sharing," in *Formal Ontology in Conceptual Analysis and Knowledge Representation*, N. Guarino and R. Poli, Eds.: Kluwer Academic Publishers, 1993.

[79] N. Guarino, "Formal Ontology, Conceptual Analysis, and Knowledge Representation," *International Journal of Human-Computer Studies*, vol. 43, pp. 625-640, 1995.

[80] N. Guarino and C. Welty, "Ontological Analysis of Taxonomic Relationships," presented at ER-2000: The 19th International Conference on Conceptual Modeling, USA, 2000.

[81] N. Guarino and C. Welty, "Towards a Methodology for Ontology-based Model Engineering," presented at ECOOP-2000 Workshop on Model Engineering, 2000.

[82] S. Guerra, M. Ryan, and A. Sernadas, "Feature-Oriented Specifications," School of Computer Science, University of Birmingham, Technical Report Nov 1996 1996.

[83] M. Halkidi, Y. Batistakis, and M. Vazirgiannis, "On Clustering Validation Techniques," *Journal of Intelligent Information Systems*, vol. 17, pp. 107-145, 2001.

[84] T. Halpin, *Conceptual Schema and Relational Database Design*, 2nd ed. Sydney, AUS: Prentice Hall, 1995.

[85] T. Halpin, "Data modeling in UML and ORM revisited," presented at International Workshop on Evaluation of Modeling Methods in Systems Analysis and Design, Heidelberg, Germany, 1999.

[86] F. Harary, R. Z. Norman, and D. Cartwright, *Structural Models: An Introduction to the Theory of Directed Graphs*. New York, NY: John Wiley and Sons, 1965.

[87] F. Harary and E. M. Palmer, *Graphical Enumeration*. New York, NY: Academic Press, 1973.

[88] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot, "STATEMATE: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transcations on Software Engineering*, vol. 16, pp. 403-413, 1990.

[89] D. M. Hilbert and D. F. Redmiles, "Extracting Usability Information from User Interface Events," *ACM Computing Surveys*, vol. 32, pp. 384-421, 2000.

[90] B. Hillier, *Space is the Machine: A Configurational Theory of Architecture*. Cambridge, UK: Cambridge University Press, 1996.

[91] B. Hillier and J. Hanson, *The Social Logic of Space*. Cambridge, UK: Cambridge University Press, 1984.

[92] D. Hix and H. R. Hartson, *Developing User Interfaces*. New York, NY: John Wiley and Sons, 1993.
[93] I. Hsi, "Analyzing the Conceptual Coherence of Computing Applications Through Ontological Excavation," Georgia Institute of Technology, Atlanta, Technical Report GIT-CC-05-07, 2004.
[94] I. Hsi, "Measuring the Conceptual Fitness of a Computing Application in a Computing Ecosystem," presented at ACM Workshop on Interdisciplinary Software Engineering Research (WISER'04), Newport Beach, CA, 2004.
[95] I. Hsi and C. Potts, "Studying the Evolution and Enhancement of Software Features," presented at Intl. Conf. Software Maintenance, San Jose, CA, 2000.
[96] I. Hsi, C. Potts, and M. Moore, "Ontological Excavation: Unearthing the Core Concepts of the Application," presented at Working Conference on Reverse Engineering, Victoria, Canada, 2003.
[97] P. Hsia, A. Davis, and D. Kung, "Status Report: Requirements Engineering," *IEEE Software*, vol. 11, pp. 12-16, 1993.
[98] X. Huang and W. Lai, "Identification of clusters in the Web graph based on link topology," presented at Seventh International Database Engineering and Applications Symposium, 2003.
[99] J. Hughes, J. O'Brien, T. Rodden, M. Rouncefield, and I. Sommerfield, "Presenting Ethnography in the Requirements Process," presented at 2nd IEEE International Symposium on Requirements Engineering, 1995.
[100] E. Hutchins, *Cognition in the Wild*. Cambridge, MA: MIT Press, 1995.
[101] S. Iacono and R. Kling, "Computerization, Office Routines, and Changes in Clerical Work," in *Computerization and Controversy: Value Conflicts and Social Choices*, R. Kling, Ed., 2nd ed. New York, NY: Academic Press, 1996, pp. 309-315.
[102] M. Jackson, *Problem Frames: Analyzing and Structuring Software Development Problems*. New York, NY: Addison-Wesley, 2001.
[103] M. A. Jackson, *System Development*. Englewood Cliffs, NJ: Prentice-Hall, 1983.
[104] E. Jacob, "Qualitative Research Traditions: A Review," *Review of Educational Research*, vol. 57, pp. 1-50, 1987.
[105] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*. Reading, MA: Addison-Wesley, 1999.
[106] A. K. Jain, M. N. Murty, and P. J. Flynn, "Data Clustering: A Review," *ACM Computing Surveys*, vol. 31, pp. 264-323, 1999.
[107] I. John, D. Muthig, P. Sody, and E. Tolzmann, "Efficient and Systematic Software Evolution Through Domain Analysis," presented at IEEE Joint International Conference on Requirements Engineering (RE'02), 2002.
[108] W. L. Johnson and M. Feather, "Building an Evolution Transformation Library," presented at 12th International Conference on Software Engineering, Nice, France, 1990.
[109] L. G. Jones and A. L. Soule, "Software Process Improvement and Product Line Practice: CMMI and the Framework for Software Product Line Practice," Carnegie Mellon, Pittsburgh, PA CMU/SEI-2002-TN-012, 2002.

[110] H. Kaindl, "An Integration of Scenarios with their Purposes in Task Modeling," presented at Designing Interactive Systems: Processes, Practices, Methods, and Techniques, Ann Arbor, MI, 1995.
[111] K. C. Kang, "Feature-Oriented Development of Applications for a Domain," presented at Fifth International Conference on Software Reuse, 1998.
[112] K. C. Kang, J. Lee, and P. Donohoe, "Feature-Oriented Product Line Engineering," *IEEE Software*, vol. 19, pp. 58-65, 2002.
[113] C. F. Kemerer and S. Slaughter, "An Empirical Approach to Studying Software Evolution," *IEEE Transactions on Software Engineering*, vol. 25, pp. 493-509, 1999.
[114] W. Kent, *Data and Reality: Basic Assumptions in Data Processing Reconsidered*. New York: North-Holland Publishing Company, 1979.
[115] H. Kohl, *From Archetype to Zetigeist*. Boston, MA: Little, Brown and Company, 1992.
[116] M. Kolberg, E. Magill, D. Marples, and S. Tsang, "Feature Interactions in Services for Internet Personal Appliances," presented at IEEE International Conference on Communications, 2002.
[117] J. Kuusela and J. Savolainen, "Requirements Engineering for Product Families," presented at 22nd International Conference on Software Engineering, Limerick, Ireland, 2000.
[118] B. Laurel, "The Art of Human-Computer Interface Design," B. Laurel, Ed. Reading, MA: Addison-Wesley, 1990.
[119] M. Lehman, "Laws of Software Evolution Revisited," presented at 5th European Workshop on Software Process Technology, Nancy, France, 1996.
[120] M. Lehman and L. Belady, *Program Evolution: Processes of Software Change*, 1st ed. Orlando: Academic Press,inc., 1985.
[121] M. M. Lehman, "Software's Future: Managing Evolution," *IEEE Software*, vol. 15, pp. 40-44, 1998.
[122] M. M. Lehman, D. E. Perry, and J. F. Ramil, "Implications of evolution metrics on software maintenance," presented at International Conference on Software Maintenance, Bethesda, MD, 1998.
[123] M. M. Lehman and J. F. Ramil, "The Impact of Feedback in the Global Software Process," presented at Workshop on Software Process Simulation and Modeling (ProSim '98), Silver Falls, OR, 1998.
[124] M. M. Lehman, J. F. Ramil, P. D. Wemick, D. E. Perry, and W. M. Turski, "Metrics and Laws of Software Evolution - The Nineties View," presented at Fourth International Software Metrics Symposium, Albuquerque, NM, 1997.
[125] T. R. Leishman and D. A. Cook, "Requirements Risks Can Drown Software Projects," *Crosstalk*, vol. 15, pp. 4-8, 2002.
[126] D. B. Lenat, "CYC: A Large-Scale Investment in Knowledge Infrastructure," *Communications of the ACM*, vol. 38, pp. 33-48, 1995.
[127] E.-P. Lim and V. Cherkassky, "Semantic Networks and Associative Databases," *IEEE Expert*, vol. 7, pp. 31-40, 1992.
[128] Y. S. Lincoln and E. G. Guba, *Naturalistic Inquiry*. London, UK: Sage Publications, 1985.

[129] D. Lowe, *PowerPoint 2000 For Windows For Dummies*. Indianapolis, IN: Wiley Publishing, Inc., 1999.
[130] Luqi, "A Graph Model for Software Evolution," *IEEE Transactions On Software Engineering*, vol. 16, pp. 917-927, 1990.
[131] A. Maccari, "Experiences in Assessing Product Family Software Architecture for Evolution," presented at 24th International Conference on Software Engineering, 2002.
[132] A. Mackenzie, A. S. Ball, and S. R. Virdee, *Instant Notes in Ecology*. Oxford, UK: BIOS Scientific Publishers Ltd, 1998.
[133] S. Maguire, *Writing Solid Code*. Redmond, WA: Microsoft Press, 1993.
[134] S. McConnell, *Code Complete*. Redmond, WA: Microsoft Press, 1993.
[135] S. McConnell, *Rapid Development*. Redmond, WA: Microsoft Press, 1996.
[136] J. McGrenere, ""Bloat": The Objective and Subjective Dimensions," presented at Computer Human Interaction 2000 (CHI 2000), 2000.
[137] J. McGrenere, "The Design and Evaluation of Multiple Interfaces: A Solution for Complex Software," in *Department of Computer Science*. Toronto, ON: University of Toronto, 2002.
[138] J. McGrenere, R. M. Baecker, and K. S. Booth, "An Evaluation of a Multiple Interface Design Solution for Bloated Software," presented at CHI 2002, Minneapolis, MN, 2001.
[139] J. McGrenere and G. Moore, "Are We All In The Same "Bloat"?," presented at Graphics Interface 2000, Montreal, 2000.
[140] D. L. McGuinness, "Conceptual Modeling for Distributed Ontology Environments," presented at The Eight International Conference on Conceptual Structures Logical, Linguistic, and Computational Issues, Darmstadt, Germany, 2000.
[141] D. L. McGuinness, R. Fikes, J. Rice, and S. Wilder, "The Chimaera Ontology Environment," presented at Seventeenth National Conference on Artificial Intelligence, Austin, Texas, 2000.
[142] D. L. McGuinness, R. Fikes, J. Rice, and S. Wilder, "An Environment for Merging and Testing Large Ontologies," presented at Seventh International Conference on Principles of Knowledge Representation and Reasoning (KR2000), Breckenridge, Colorado, 2000.
[143] A. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," presented at Tenth Working Conference on Reverse Engineering, Victoria, BC Canada, 2003.
[144] T. Mens and S. Demeyer, "Future Trends in Software Evolution Metrics," presented at 4th International Workshop on Principles of Software Evolution, Vienna, Austria, 2002.
[145] M. Moore, "A Survey of Representations for Recovering User Interface Specifications in Reengineering," College of Computing, Georgia Institute of Technology, Atlanta, GA July 16, 1996 1996.
[146] M. Moore, "User Interface Reengineering," in *College of Computing*. Atlanta, GA: Georgia Institute of Technology., 1998.
[147] B. A. Myers, "User Interface Software Tools," *ACM Transactions on Computer-Human Interaction*, vol. 2, pp. 64-103, 1995.

[148] B. A. Nardi, "Studying Context: A Comparison of Activity Theory, Situated Action Models, and Distributed Cognition," in *Context and Consciousness: Activity Theory and Human-Computer Interaction*, B. A. Nardi, Ed. Cambridge, MA: MIT Press, 1996, pp. 69-102.
[149] B. A. Nardi and V. L. O'Day, *Information Ecologies: Using Technology with Heart*, Reprint Edition ed. Cambridge, MA: MIT Press, 2000.
[150] J. Nielsen, *Usability Engineering*. Cambridge, MA: Academic Press, 1993.
[151] G. M. Nijssen and T. A. Halpin, *Conceptual Schema and Relational Database Design*. New York: Prentice Hall, 1989.
[152] N. J. Nilsson, *Principles of Artifical Intelligence*. Palo Alto, CA: Tioga Publishing Company, 1980.
[153] D. Norman, *The Invisible Computer*. Cambridge, MA: MIT Press, 1998.
[154] D. A. Norman, *The Design of Everyday Things*. New York, NY: Doubleday, 1988.
[155] P. Oreizy, "A Flexible Approach to Decentralized Software Evolution," presented at 1999 International Conference on Software Engineering, Los Angeles, CA, 1999.
[156] P. Oreizy, M. M. Gorlick, R. N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum, and A. L. Worl, "An Architecture-based Approach to Self-adaptive Software," *IEEE Intelligent Systems*, vol. 14, pp. 54-62, 1999.
[157] D. E. Perry, "Dimensions of Software Evolution," presented at International Conference on Software Maintenance, Victoria, BC Canada, 1994.
[158] H. Petroski, *The Book on the Book*. New York: Alfred A. Knopf, 1999.
[159] H. Petroski, *Design Paradigms: Case Histories of Error and Judgment in Engineering*. Cambridge: Cambridge University Press, 1994.
[160] H. Petroski, *The Evolution of Useful Things*, 1 ed. New York: Vintage Books, 1992.
[161] H. Petroski, *Invention by Design: How Engineers Get From Thought to Thing*. Cambridge, MA: Harvard University Press, 1996.
[162] H. Petroski, *The Pencil: A History of Design and Circumstance*. New York, NY: Alfred A. Knopf, 1992.
[163] C. Potts, "Requirements Models in Context," presented at 3rd International Symposium on Requirements Engineering (RE'97), Annapolis, MD, 1997.
[164] C. Potts, "Using Schematic Scenarios to Understand User Needs," presented at Designing Interactive Systems: Processes, Practices, Methods, and Techniques, Ann Arbor, MI, 1995.
[165] C. Potts, A. Anton, and K. Takahashi, "Inquiry-Based Requirements Analysis," in *IEEE Software*, vol. 2, 1994, pp. 21-32.
[166] C. Potts and I. Hsi, "Abstraction and context in requirements engineering: Toward a Synthesis," *Annals of Software Engineering*, vol. 3, pp. 23-61, 1997.
[167] R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 4th ed. New York, NY: McGraw Hill, 1997.
[168] M. E. Raichle, "Visualizing the Mind," *Scientific American*, vol. 270, pp. 58-64, 1994.

[169] E. S. Raymond, *The Cathedral and the Bazaar*. Sebastopol, CA: O'Reilly and Associates, 1999.
[170] D. Richardson, *MAGE*. Duke, NC: Biochemistry Dept., Duke University, 2002.
[171] E. M. Rogers, *Diffusion of Innovation*, 4th ed. New York: The Free Press, 1995.
[172] Y. Rubinsky, *SGML on the WEB: Small Steps Beyond HTML*. New Jersey: Prentice-Hall, 1995.
[173] S. Rugaber, "Program Comprehension," *Encyclopedia of Computer Science and Technology*, vol. 35, pp. 341-368, 1995.
[174] S. Rugaber and M. Guzdial, "Ectropic Software," presented at Workshop on Software Change and Evolution, Los Angeles, CA, 1999.
[175] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, *Object-oriented Modeling and Design*. Englewood Cliffs, NJ: Prentice-Hall, 1991.
[176] R. Sedgewick, *Algorithms in C: Part 5*, 3rd ed. New York, NY: Addison-Wesley, 2002.
[177] E. E. Smith and D. L. Medin, *Categories and Concepts*. Cambridge, MA: Harvard University Press, 1981.
[178] I. Sommerville, T. Rodden, P. Sawyer, R. Bentley, and M. Twidale, "Integrating Ethnography Into The Requirements Engineering Process," presented at IEEE International Symposium on Requirements Engineering, San Diego, CA, 1992.
[179] I. Sommerville and P. Sawyer, *Requirements Engineering: A Good Practice Guide*. New York, NY: John Wiley and Sons, 1997.
[180] J. F. Sowa, *Conceptual Structures: Information Processing in Mind and Machine*. Reading, MA: Addison-Wesley, 1984.
[181] J. P. Spradley, *The Ethnographic Interview*. New York, NY: Harcourt Brace Jovanovich College Publishers, 1979.
[182] E. Stroulia, M. El-Ramly, and P. Sorenson, "From Legacy to Web through Interaction Modeling," presented at International Conference on Software Maintenance, Montréal, Canada, 2002.
[183] E. Stroulia and R. V. Kapoor, "Reverse Engineering Interaction Plans for Legacy Interface Migration," presented at 4th International Conference on Computer Aided Design of User Interfaces, Valenciennes, France, 2002.
[184] Sun Microsystems,"A Visual Index to the Swing Components,"From, available at http://java.sun.com/docs/books/tutorial/uiswing/components/components.html, 2004.
[185] A. Sutcliffe, Maiden, Minocha, and Manuel, "Supporting Scenario-based Requirements Engineering," *IEEE Transactions on Software Engineering*, vol. 24, pp. 1072-1088, 1998.
[186] S. M. Sutton, Jr. and L. J. Osterweil, "Product Families and Process Families," presented at 10th International Software Process Workshop, Dijon, France, 1996.
[187] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf, "A Conceptual Basis for Feature Engineering," *Journal of Systems and Software*, vol. 49, pp. 3-15, 1999.
[188] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf, "Feature Engineering," presented at Ninth International Workshop on Software Specification and Design, 1998.

[189] A. van Duresen and T. Kuipers, "Identifying Objects Using Cluster and Concept Analysis," presented at International Conference on Software Engineering, Los Angeles, 1999.
[190] D. K. Van Duyne, J. A. Landay, and J. Hong, *The Design of Sites*. New York, NY: Addison-Wesley, 2003.
[191] G. M. A. Verheijen and J. Van Bekkum, "NIAM: An Information Analysis Method," in *Information Systems Design Methodologies*, T. W. Olle, H. G. Sol, and A. A. Verrijn-Stuart, Eds. Amsterdam: North-Holland Publishing Company, 1982.
[192] Y. Wand, V. C. Storey, and R. Weber, "An Ontological Analysis of the Relationship Construct in Conceptual Modeling," *ACM Transactions on Database Systems*, vol. 24, pp. 494-528, 1999.
[193] Y. Wand and R. Y. Wang, "Anchoring Data Quality Dimensions in Ontological Foundations," *Communications of the ACM*, vol. 39, pp. 86-95, 1996.
[194] S. Wasserman and K. Faust, *Social Network Analysis*. Cambridge: Cambridge University Press, 1994.
[195] T. F. E. Wikipedia,"Cathedral of Chartres,"From, available at http://en.wikipedia.org/wiki/Cathedral_of_Chartres, 2005.
[196] T. F. E. Wikipedia,"Notre-Dame de Reims,"From, available at http://en.wikipedia.org/wiki/Notre-Dame_de_Reims, 2005.
[197] N. Wirth, "A Plea for Lean Software," *IEEE Computer*, vol. 28, pp. 64-68, 1995.
[198] P. Zave, "Feature Interactions and Formal Specifications in Telecommunications," *Computer*, vol. 26, pp. 20-28, 30, 1993.